A SIMULATOR FOR MICROCODED SIGNAL PROCESSORS

A Thesis Submitted

In Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

By

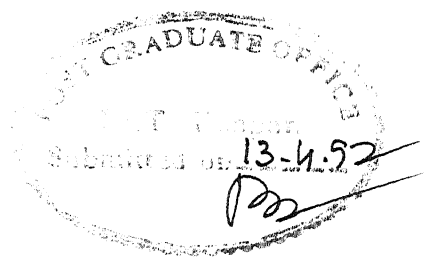
J. VISWESWAR

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1992



CERTIFICATE

It is certified that the work contained in the thesis entitled " A Simulator for Microcoded Signal Processors " by Mr. J Visweswar, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Anil Mahanta

(Dr. Anil Mahanta)
Assistant Professor,
Department of Electrical Engineering
Indian Institute of Technology,
Kanpur.

April, 1992.

EE-1992-M-VIS-SIM

ACKNOWLEDGEMENTS

It is a great pleasure in expressing my sense of gratitude to all those who helped me in completing this thesis.

I first thank Dr A.Mahanta whose guidence helped me a lot.He gave me constant encouragement and help during my thesis.

The discussions with D.Venkaih helped me very much. I am very grateful to M.K.V.S.Rao,Sudhakar,Lakshmi Prasad, C.N.Sankar, Dandam, Ravichander,Perraju,Umakant and all those who made my stay in IIT, Kanpur a memorable one.

Contents

ABSTRACT

Microcoded systems are used extensively in signal processing applications. They attain high functional parallelism and thereby achieve high throughput.Simulation of a system is a software tool widely used for the system development and checking. In this thesis, a number of microcoded signal processors are proposed and a simulator for these systems is developed. The simulator permits programs to be finely-tuned in software prior to making significant effort to bring up the target system in hardware. The simulator is interactive, it allows display of registers and memory contents. Single and full-speed run is possible. Using this simulator performance of the proposed microcoded systems is evaluated with a few DSP algorithms.

CHAPTER-1

INTRODUCTION

1.1 Digital Signal Processing (DSP):

Digital signal processing is concerned with representation of signals by sequence of numbers and processing these sequences. The purpose of such processing may be to estimate characterstic parameters of a signal or to transform a signal into a form which is, in some sense, more desirable than the original. Signal processing finds wide applications in various diverse fields such as biomedical engineering, acoustics, sonar, radar, seismology data and speech communication, nuclear sciences and many other areas.

1.2 Approaches to Implementation of Signal Processing Algorithms

Efficient implementation of DSP algorithms require high speed algebraic manipulation of data involving mainly multiplication and addition. The computational requirements of DSP application range from a few hundred operations per second to several hundreds of millions of operations per second(Mops). A single, fixed implementation scheme will clearly, be non-optimal vis-a-vis cost and performance index. Besides cost and speed, another important criteria is the acuracy of implementation. Depending on the particular application or class of application, tradeoff exists between:

— speed

— accuracy

— cost (both hardware and programming)

While software implementation on a general purpose micro processor is often cost effective in non real-time and/or low data rate applications, this approach soon proves to be inadequate as the amount of data and/or the data-rate increases, and a hardware solution becomes a necessity.

The various hardware solutions can be broadly categorized into the following types:

— General purpose micro processor with coprocessor,

— Programmable DSP microprocessor,

— Microprogrammable Processors.

The first approach is a modest extension of the conventional microprocessor, and is satisfactory for many low-to-moderate data-rate applications.

Programmable single chip digital signal processor is a microprocessor whose architecture is optimised to process sampled data at high rates. These architecture continue to be adequate for many medium to moderately high throughput applications. Existing DSP micros like TMS320, ADSP2100, DSP56000, WEDSP16 & TS68930 etc [spectrum 87] belong to this category.

The third approach is the most efficient, but expensive solution for throughput requirements in DSP. The processors are implemented using multiple hardware funtional units under microprogram control, which allows the designer increased control over the system

architecture to meet very high throughput requirements. These architectures are generally called as microcoded systems.

1.3 Microcoded Signal Processors (MCSP)

A microcoded system employs building block IC's to construct a signal processor. A comprehensive set of building blocks are available to enable high performance digital signal processors to be constructed.

The basic difference in the design philosophy between microprocessor circuits and microcoded circuits is that the functions which are integrated on to a single microprocessor device are partitioned as seperate devices in a microcoded circuit. Microcoded circuits offers a high degree of functional parallelism where many operations can be performed simulteneously unlike in a conventional microprocessor thereby increasing the throughput.

Since each device in a microcoded system can operate independently for a synchronous operation, a proper coordination is needed. This is achieved by having a single instruction which is synchronous to a system clock. Each instruction is called a microinstruction. Each microinstruction contains the control bits for the various devices in the system at different locations called `fields'. That is, each field direclty controls the corresponding device of a microcoded system. A sequence of microinstructions is called a microprogram. Microprograms are written and stored in a memory called control store(microcode memory).

During each system clock cycle, a microcode memory location is accessed and the microcode residing at that location is supplied to

the microcoded devices in the system. The width of the microcode memory depends on the number of devices in the system. The power of the microcoded processor lies in the fact that during each clock cycle, each component can execute an instruction, allowing the system to attain high throughput. Such a processor is invariably attached to a host computer. One such attached processor is shown in Fig 1.1. Host machine could be a personel computer, mini or a main computer.

1.4. Objectives and Scope of the Thesis

In this thesis an attempt has been made to develop a simulator for a MCSP system for developing and executing various signal processing algorithms. The objectives of the current work are described below:

(i) To define various MCSP architectures based on performance and speed which will work as an attached processor to an external host.

(ii) To develop a simulator for the architectures using ADSP-14** and 11** chip set.

(iii) To test the simulator through execution of some simple microprograms.

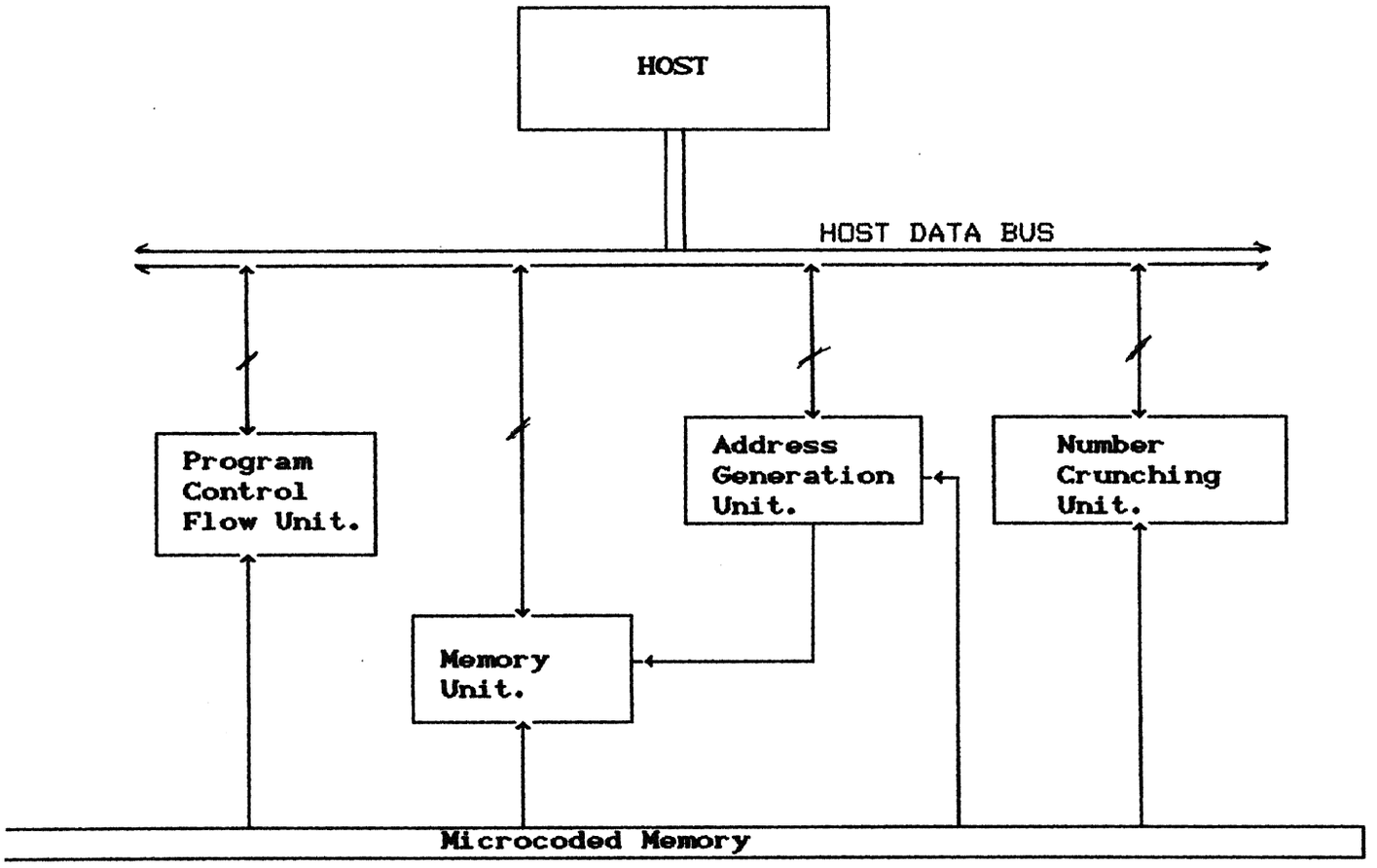


Fig 1.1 HOST attached Microcoded System

1.5 Organisation of the Thesis

In Chapter 2 various MCSP architectures are proposed and the various functional elements used are briefly described.

The working of the simulator for the three MCSP architectures is explained in Chapter 3.

In Chapter 4 algorithms like 1D-convolution and matrix multiplication are tested on the MCSP systems and their performance is evaluated. The meta-assembler used is also explained.

Finally in Chapter 5, we conclude the thesis with suggestions for further work.

CHAPTER 2

MICROCODED SIGNAL PROCESSORS : AN OVERVIEW

In this Chapter various microcoded architectures are proposed,and discussed. The systems are simulated on HP-9000 and tested with test algorithms.

2.1 System 1 : Uniprocessor with LOCAL MEMORY

The block-diagram of this system is shown in Fig 2.1. The system can be devided into three main sections :

(i) Control section,
(ii) Address generation section ,
and (iii) Number crunching section .

2.1.1 *Control Section :*

The main components of the control section are the program sequencer , the microcoded memory and the pipeline register.

The Program Sequencer

The microprogram sequencer's main task is to provide the appropriate microprogram addressing to support programming requirements (e.g.,looping , jumping , branching subroutines , conditional testing and interrupts). During each micro-instruction , the program sequencer monitors the conditions and instruction to determine the address of the next microinstruction thereby controlling

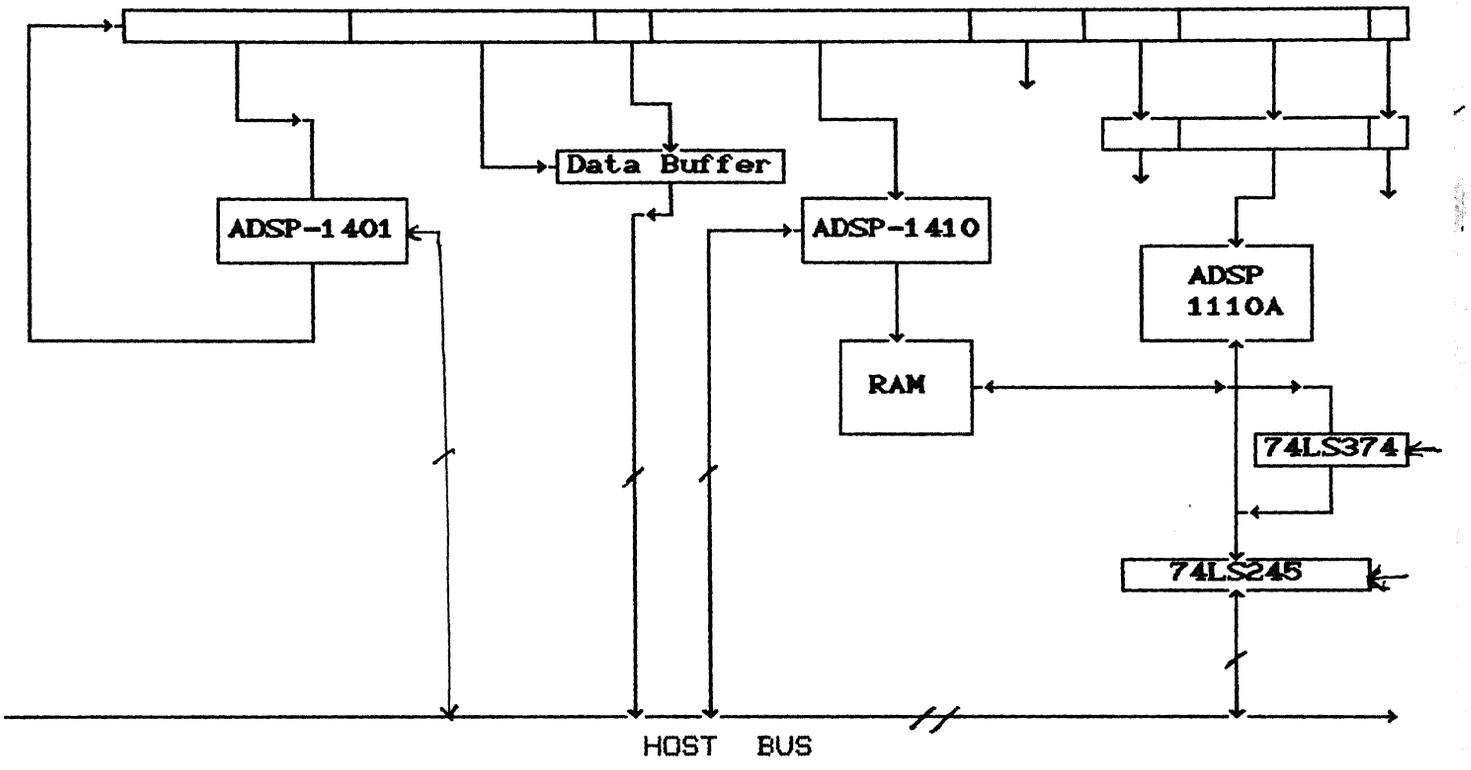


Fig 2.1 SYSTEM 1 : System with Uniprocessor and Local Memory.

the program flow . The microprogram memory which contains the microinstructions, may be a RAM or a ROM . The program communication between the microprogram memory and program sequencer is shown in Fig 2.2.

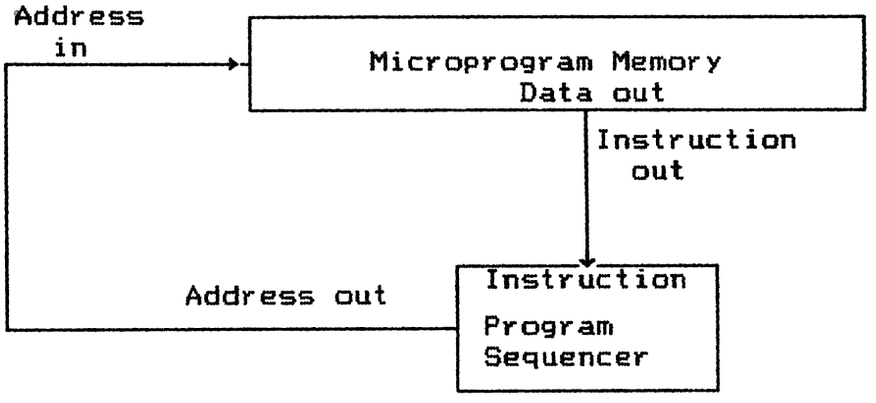


Fig 2.2 : Genetation of Next Address

In a complex high performance system the instruction in the microcode memory will not be fetched sequentially as previously described. In most cases,sophisticated program flow will be needed . As with other devices in the system , the program sequencer will receive its instruction from microcoded memory as explain earlier . In essence , the microcoded memory is instructing the program squencer to get the next instruction . Fig 2.3 shows an example of a non-sequential program flow. Each instruction gives the sequencer the information on how to get the next instruction to be executed.

Current Micro-program address	Program Sequencer Instruction	Next Micropr-ogram address
10	Jump to location 20	20
20	Jump to subroutine	50
50	Execute instruction	:
.	.	.
	Return from subrou-tine	21
21	_____	—

Fig 2.3 : Non-sequential Program Flow

These jumps and conditional calls can also be made conditional upon some external event . The program sequencer generates address of the instruction which determines program flow in several different ways , three of which are shown in Fig 2.4 .

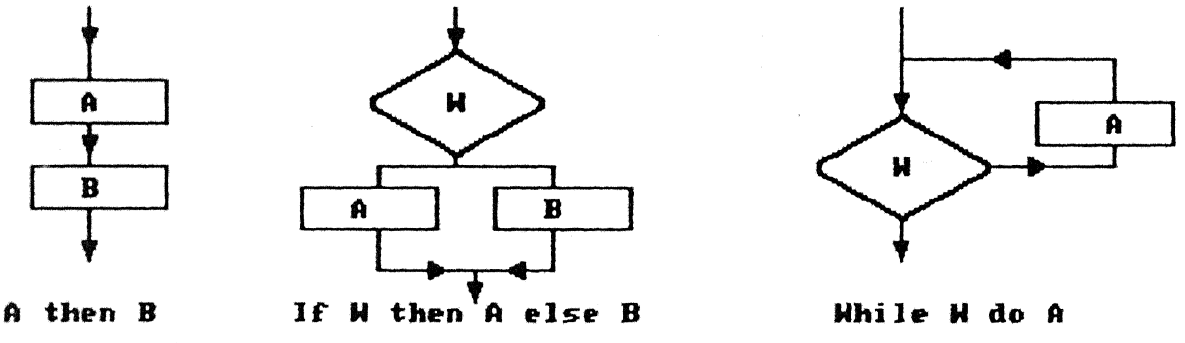


Fig 2.4 : Examples of Program Flow Charts

Program sequencer removes overhead by Keeping track of program sequencer incrementing, soubroutine handling and and outside-world interrupts.The sequencer thus has the following main functions:

1) handle normal program flow, incrementing the program counter

by one in each cycle,

2) keep track of subroutine addressing and manage the return address stack,

3) manage loops in an overhead-free manner controlled by on-chip loop counters,

4) jump to appropriate exception handler routines when data overflow is detected,

5) service interrupts from external I/O devices, jumping to appropriate interrupt service routines.

In order to accomplish these functions a modern program sequencer includes the folloowing components :

- Program counter
- conditional logic tester
- stack for
  - subroutine and interrupt return addresses,
  - counter values
  - jump addresses
  - loop counters

Pipeline latches

Microcode memories are constructed with either ROM or RAM components . Depending on the depth and width of microcodememory needed , many memory chips will need to be cascaded .The accessing of a memory involves an address being stable , the location contents being accessed and data output becoming stable. Between memory accesses

as the address and data become stable , the output of the memory is in an undetermined state . The consequences of having these undetermined bits driving the devices of the microcode circuit is that unpredictable circuit operation is imminent . To prevent such operation , pipeline latches are required . A D-tyoe latch is placed at the output of the microcode memory and is clocked at every rising edge of the system clock as shown in Fig 2.5,

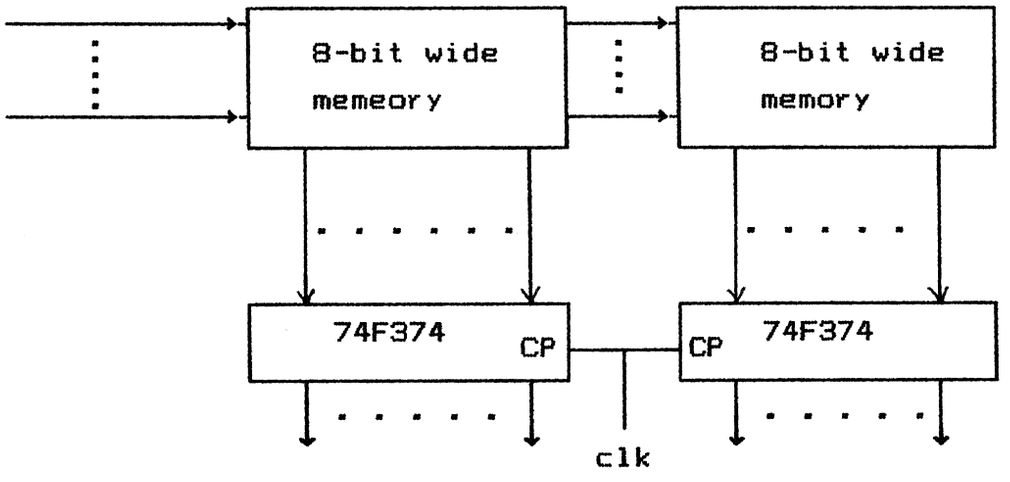


Fig 2.5 : Pipeline Latches

The timing of the microcode memory is such that valid output exists prior to the rising edge of the clock and stable information is loaded into the latch . It will hold this microcode instruction until the next clock edge . During this time a new memory location can be accessed . The process of holding a valid instruction while the next is being fetched is called pipelining , hence the name pipeline latch .

The functional unit used as control unit is ADSP-1401 .The ADSP-1401 is a high speed microprogram sequencer ,optimised for demanding sequencing tasks found in DSP and general purpose computers.

ts features such as on chip storage and control of ten prioritised nd maskable interrupts , four decrementing event counters,absolute elative and indirect addressing capability and a dynamically :onfigurable 64 word RAM ideally meet the sequencing demands of any microcode circuits The ADSP-1401 contains an internal pipeline register and canbe connected directly to the output of the microcode memory . No externl pipeline latch is required as in the other devices. A detailed description of ADSP-1401 is given in **Appendix A.**

2.1.2 *Address generation Section*

Addressing complexity for signal processors can range from integer counting to the more complex sequence of data pointing occuring in FFTs. In signal procesing there are more memory accesses than there are data points, but the memory access sequence although long ,is very well structured and makes possible the design and the use of dedicated address generators. The FFT is a good example of an algorithm that can use an address generator very effectively. The use of address generator relieves the CPU from the mundane task of address generation, and permits the CPU to use all its machine cycles for arithmetic computations,thus boosting the overall system performance.

Desirable function of an address generators include:

1) send a precomputed address to data memory,
2) modify the address by an offset value,
3) perform logical operations and shifts,
4) compare a pointer to a preset value,
5) reset pointer to buffer origin,
6) reverse the order of bits (needed in FFT)

In the general purpose system explained here, a 16-bit data

bus structure is used, Data operands must reside in a data memory and all data memories and buses are 16-bit wide.The length of data memory is determined by maximum size of any data record that must be stored. Here a 1K word memory is chosen. But this length can be increased as desired using architecture definiton file.

The functional unit, chosen here , which includes all the above features of address generation is ADSP-1410. The ADSP-1410 is a fast, flexible address generator which rapidly generates the data memory addresses required for a wide range of digital signal/array procesors and other high performance computers. The ADSP-1410's architecture features a 16-bit ALU, a comparator and a 30 16-bit registers. In a single cycle, the device can output a 16-bit memory address, modify this memory address, and detect when the address value has moved to or beyond a pre-set boundary and conditionally loop back to the top of a circular buffer. Consequently, circular buffers and modulo addressing for data memories can be implemented without overhead.

The ADSP-1410 contains an internal microcode pipeline latch and can be connected directly to the output of the microcode memory. No external pipeline latch is required. A detailed desription of ADSP-1410 chip and it's working is given in **Appendix B.**

Data Buffer

If the data values are to be specified directly from the microprogram instruction, a constant data field is included in the instruction field. Since the data field cannot drive the data bus directly, it is loaded into a data buffer. In order to control the data buffer one control bit is included in the microprogram

instruction field. Whenever it is high, buffer is enabled and data bus is loaded with constant data field of the micrioprogram instruction. This is very important while initialising various functional units of microcoded systsem.

2.1.3 Number Crunching Section

In order to perform high speed arithmetic computations, an appropriate arithm@tic device or devices are to be used. The device should have the desired port structure, high computation speed and overall features. In the present work ADSP-1110A is chosen for the purpose.

ADSP-1110A is a high speed, low power, single-port 16x16-bit multiplier/accumulator (MAC) offering unique advantages such as comapact 28-pin DIP package, simple system interface to single bus peripherals, reduced cost and featues such as overflow and saturation. This device will perform multiplications, additions and subtractions which are the core operations in most algorithms.It offers mixed mode multipliers such as multiply/subtact.The chip details and its working are given in **Appendix C.** Since ADSP-1110A has no internal pipeline latch, an external latch is used to drive the multiplier/accumulator instruction bits from microcode instruction memory.

74LS374 is an output pipeline latch used to latch the output data whenever ADSP-1110A executes an output instruction. One control bit is included in the microinstruction to control latch operations.Its operation is shown in Fig 2.6.

Functional Table			
o/p con	clock	D	output
L	↑	H	H
L	↑	L	L
L	L	X	Q_o
H	X	X	Z

Fig 2.6 Functional Table of 74LS374

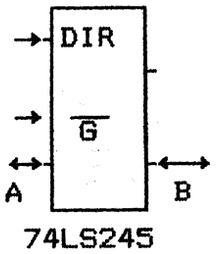
Bidirectional Buffer

The MAC-unit is interfaced to the host data bus through a bidirectional buffer (74LS245). This is to avoid bus-contention when more than one MAC unit is connected to the data-bus(see Fig 2.9 & 2.10). Two control bits are provided in the microinstruction to control this functional device. It's operation is shown in Fig 2.7,

control bits	operation
11	Read from data bus
10	Write onto data bus
00	Tristated

Fig 2.7 : Control of Bidirectional Buffer

The functional table of 74LS245 is shown in Fig 2.8,



Functional Table		
Enable $\overline{G}$	DIR control	Operation
L	L	B data to A bus
L	H	A data to B bus
H	X	Isolation

Fig 2.8 : Functional Table of 74LS245

2.1.4 *Microcode Memory Width*

The number of control bits required by the various functional units of the system of Fig 2.1 are listed below,

1)Program Sequencer	7
2)Data Field	16
3)Data buffer contro	1
4)Data memory control	2
5)Address Generator Field	12
6)MAC field	8
7)MAC o/p latch control	1
8)Bidirectional buffer	2
	49

Thus 49 microcode bits are required to drive the system components.

2.2 SYSTEM 2: System with Two Processors and A Shared Memory

One of the advantages of a microcoded systems is that many arithmetic devices can be organised to function in parallel to increase the throughput of the system. Fig 2.9 shows an architecture having two MAC units sharing one memory unit via an internal data bus.This internal data bus is separated from the host data bus by a bidirectional buffer 74LS245. The MACs in tern are connected to the

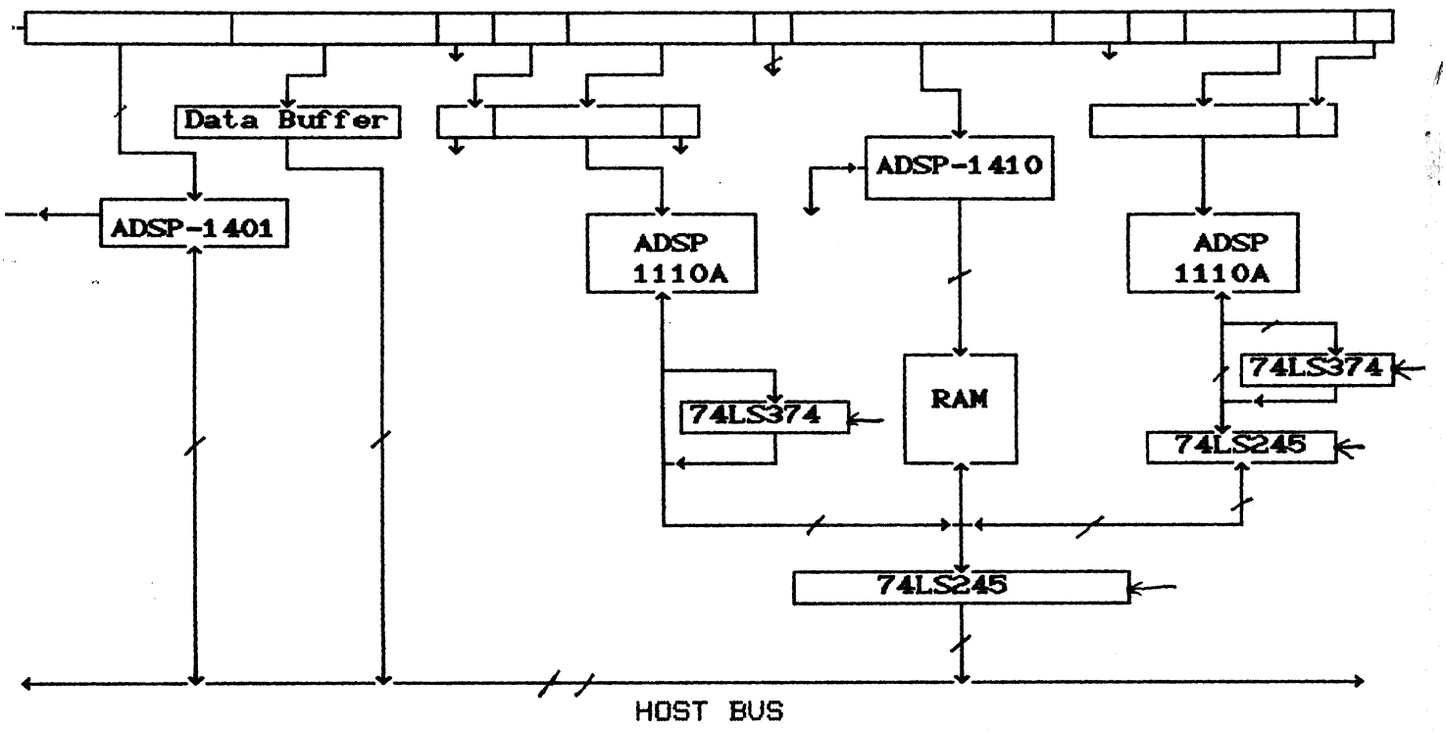


Fig 2.9 SYSTEM 2 : System with Two Processors and A Shared Memory

internal data-bus via tri-state buffer (74LS245). This is to ensure that both MACs do not try to write data-memory simultaneously. The incluson of the second MAC with its glue-logic increases the microcode width to 49+11 = 60 (MAC #2 needs 8, MAC o/p latch needs 1 and tri-state buffer needs 2).

2.3 SYSTEM 3 : Sytsem with Two Processors and Two Memories

In System 2 both the processors cannot access different data items in the memory simultaneously, because of common-bus structure, one of them must wait till the other has accessed the data. This seriously limits the throughput of the system even though the system has two processors. The architecture of Fig 2.10 attempts to overcome this limitation by providing separate data memories for the two MAC units. This architecture is essentially a duplication of the unit of Fig 2.1, with the two units now connected to host data bus via tri-state bidirectional buffers. The microcode width now becomes 74 bits.

In the next chapter the simulation of the architecture described here discussed and in chapter 4 their performance is evaluated with a few examples.

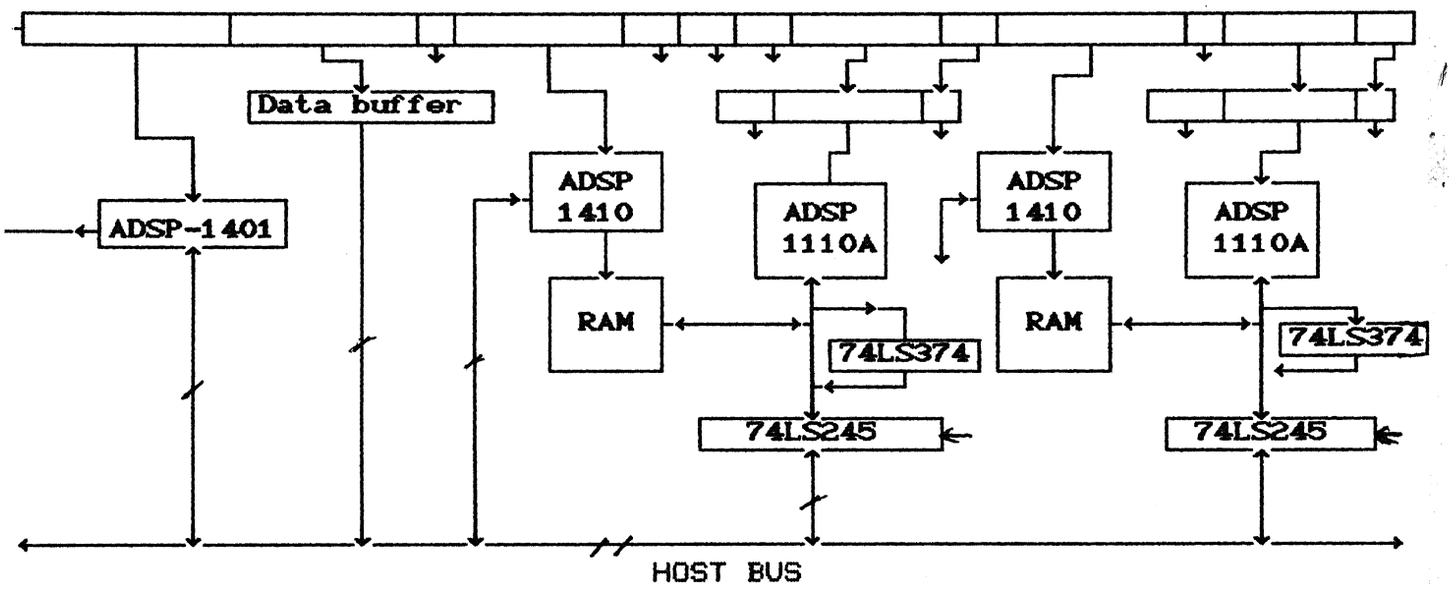


Fig 2.10 SYSTEM 3 : System with Two Processors and Two Memories

CHAPTER 3

SIMULATOR

3.1 Introduction

A simulator is a software tool which aids the user in program development and checking. The Simulator permits programs to be fine-tuned in software prior to making the significant effort to bring up the target system in hardware. The Simulator acts as much like the target system as the nature of software permits. Programs under development are run on a host machine. For simulating input/output, physical I/O devices are replaced by host memory locations. Breakpoints can readily be set so that when a program arrives at a specified program or data location, there is a display of all register contents when the break point occured or Trace of system activity before the breakpoint. Sections that cause a crash or dead-ended loops can readily be identified. The host's memory plays the role of memory in the target system, acting as ROM when so specified. Even interrupts can be simulated by timers set to trigger a simulated interrupt line.

3.2 Simulator Blocks

The main building blocks of the simulator are the simulation of,

(i) sequencer (ADSP 1401)

(ii) address generator (ADSP 1410)

(iii) arithmetic unit (ADSP 1110A)

For simulation of the seuqencer and the address generator we ollowed the approach given by WANDHEKAR [2]. In what follows, we ive a brief descripton of the simulation methodology for these units.

1.2.1 *Simulation of the Sequencer* (ADSP-1401)

The details of the chip are given in the Appendix -A. All the features of the chip are simulated. Each of the sequencer instruction is simulated using a separate routine. Registers and signals are expressed in the form of global variables. The simulation process is as follows.

In an instruction cycle, first the subroutine for the instruction corresponding to the opcode presented to the sequencer is executed. Then the program checks for an interrupt request. If a request is pending the program counter is loaded with interrupt vector. At the end, the program checks the interrupt pins and interrupt sources for an interrupt. If any of these sources is active a request is raised. The working of the sequencer is as shown in the flow chart of Fig 3.1.

The ADSP-1401 processes eight external and two internal interrupts. The two internal interrupts are reserved for stack overflow(IR9) and counter underflow(IR0). The simulator has internal cycle counters, which can be set to model interrupt devices. User can

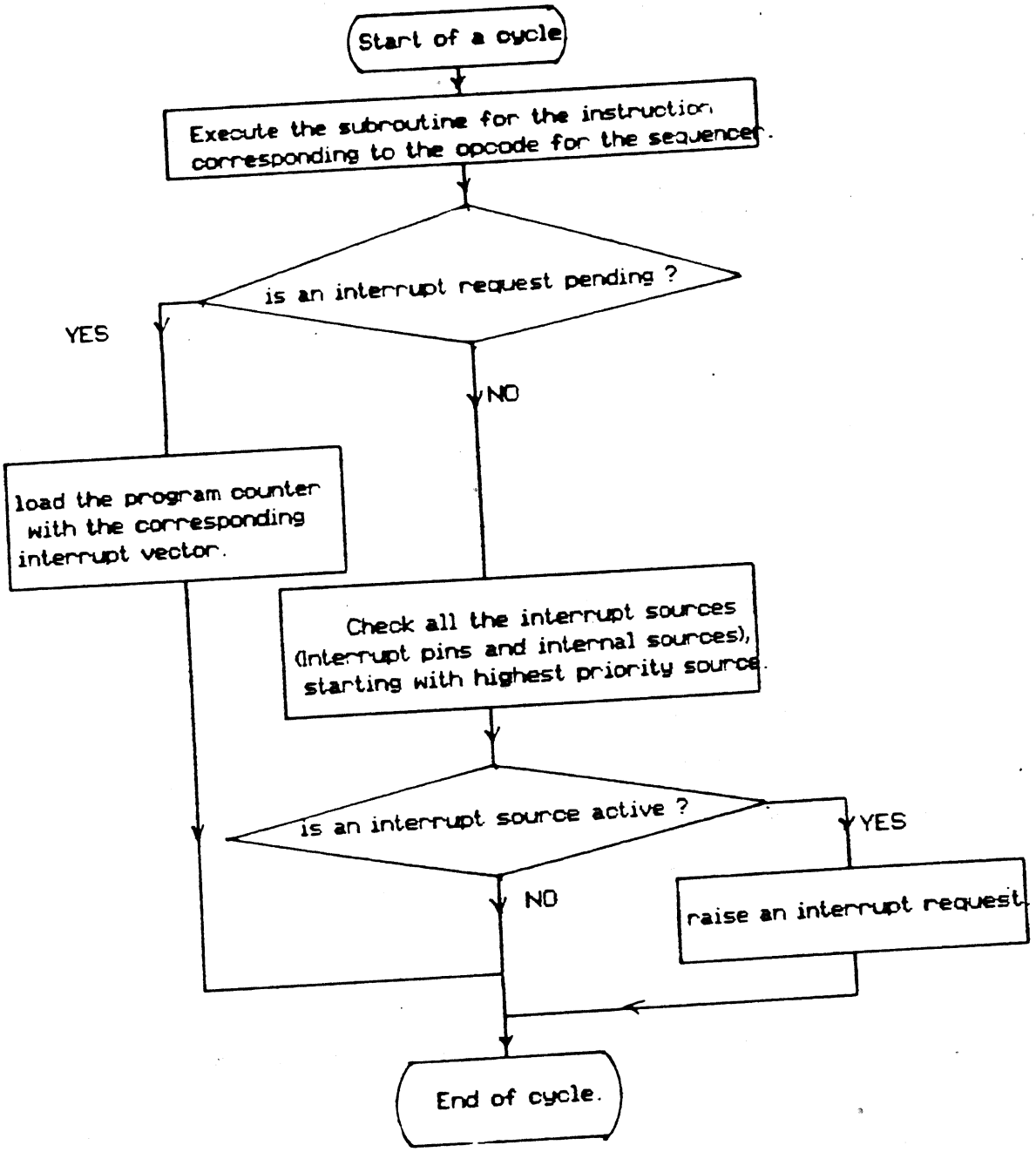


Fig 3.1 Simulation of the Sequencer ADSP-1401.

activate any of the eight external interrupting sources (IR1-IR8) by specifying time interval in cycles between two consecutive interrupts. When internal time expires, an interrupt is issued at the corresponding level. To disable the interrupt, the interrupt period is set to zero.

The other major features of the sequencer simulation are ,

(i) Internal 64-word RAM implementing two distinct stacks,a subroutine stack and a register stack, when stack overflow is detected, the interrupt IR9 is raised.

(ii) Four independent counters are used for maintaining loops and event tracking.

(iii) When the sign bit of the status register is set, IRO is raised.

3.2.2 *Simulation of the Address Generator* [ADSP-1410]

The simulation process is similar to the sequencer. In every cycle the subroutine for the instruction corresponding to the opcode for address generator is executed. The instruction set and details of address generator chip are given in **Appendix-B.**

3.2.3 *Simulation of the Arithmetic unit* [ADSP-1110A]

The simulation of Arthmetic Unit is also similar to sequencer and address generator. The instruction set and details of the chip are given in **Appendix-C.**

3.3 The Working of The Simulator

As shown in Fig 3.2., the simulator reads the architecture description file, object code file and symbol table file output by the meta-assembler. An architecture description file is input to simulator to start simulator session. The object code file is loaded interactively . The symbol table file is loaded implicitly when the object file is loaded. The definition file is loaded implicitly when the simulator is run.

By reading the debug symbol table (created by the meta-assembler), the simulator interacts with the user symbolically. The user can make references to the variables and program labels using symbols defined in the user program avoiding the need to decode the symbols. The simulator can disassemble the microinstruction, making full use of the symbols defined in the user program.

The simulator is interactive. By reading the architecture file, it configures itself with the target system architecture. The user can download the data memory from the terminal. The user can also upload the data memory contents into a file for subsequent analysis. The user can also observe the functioning of various components of the system by displaying registers/memory contents of sequencer, address generator, arthmetic unit, microprogram memory and the data memory.

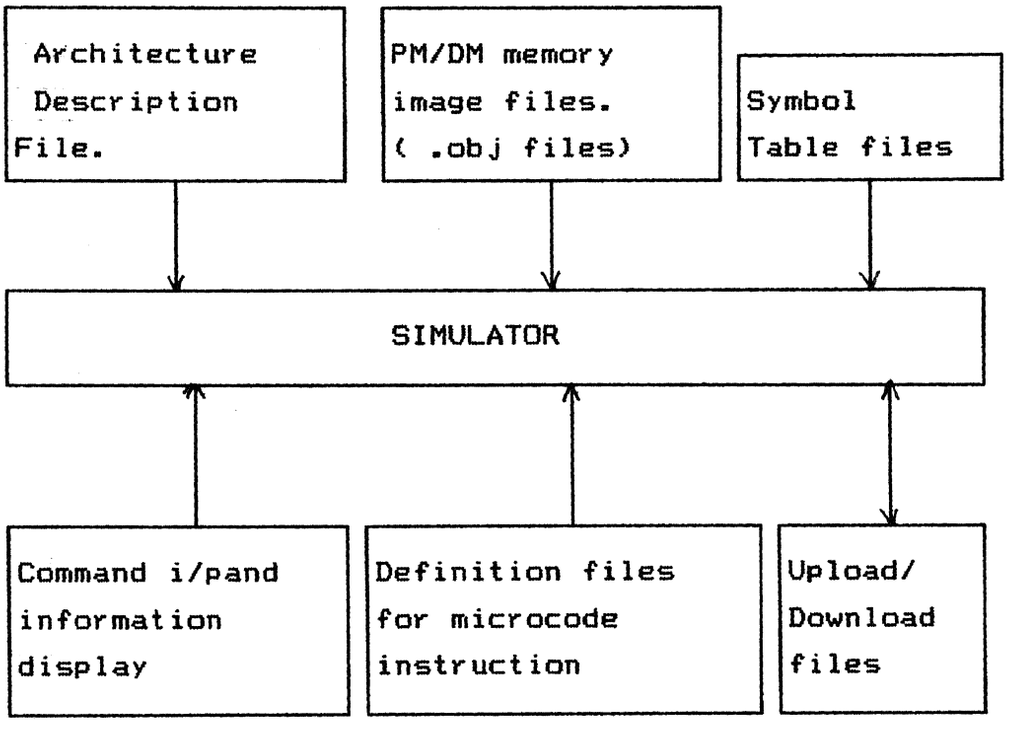


Fig 3.2 Simulator inputs and outputs.

A user working in HP-9000, should first compile the simulator and create a .out file. A simulator session is started by executing the .out file. When the user is working on PC, he should first compile the simulator in Turbo-C and create an executable file for the simulator. By typing MSIM <> user starts a simulator session. The main operation of the simulator is shown in Fig 3.3.

As explained earlier the inputs to the simulator are the architecture description file and the definition file. These files are explained below.

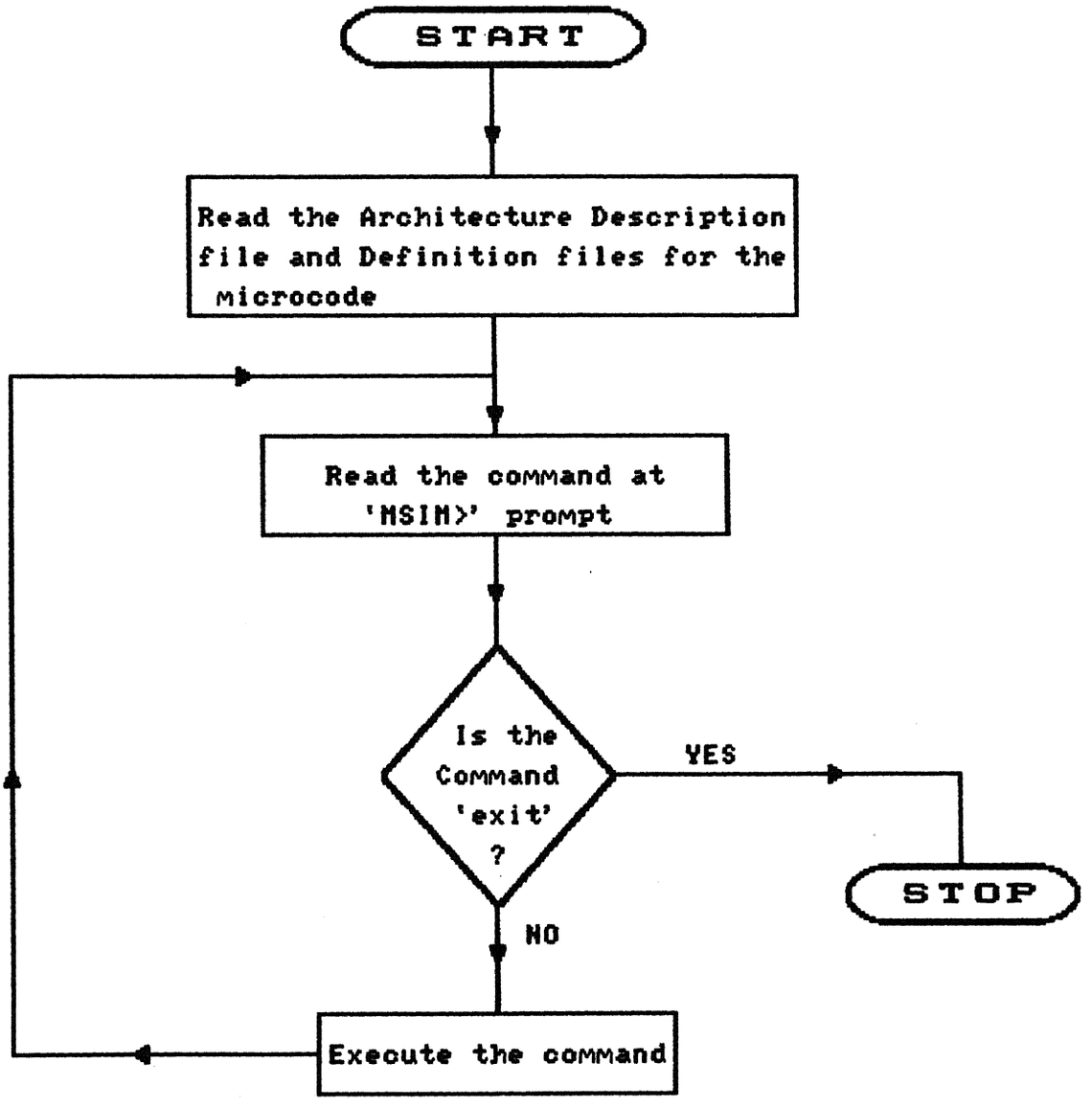


Fig 3.3 The main program flow of the Simulator.

Architecture Description File

A valid architecture description file starts with '@' and ends with '@'. In the architecture description file user can specify the type of architecture he needs, i.e. he can specify the length of the data memory and program memory. If any one of the above parameters is not specified, a minimum default value is assumed. For example if user wants the following specification,

D-mem length 2000

p-mem length 3000

he has to input the above specifications as follows:

```
@
dmem  2000
pmem  3000
@
```

Definition Files

The simulator assumes the microinstruction as a set of fields. The information about each field, i.e. field number, length of field, beginning and end of each field etc. is read by the simulator from definition files. These files are created by the meta-assembler.

3.4 Simulator Commands

After collecting architecture description file and definition file, the simulator allocates memory to various components of the system, i.e. sequencer, address generator, data memory, program memory etc.

The simulator then prints the simulator prompt,MSIM>, and waits for commands.

The simulator executes various commands present in a command table. When a command is given, it searches the command table and executes that particular command. The user can find the various commands used by the simulator, by means of **help** command.

3.4.1 *Loading Data Memory and Program Memory*

In order to execute a microprogram, the program memory of the simulator is to be loaded with the microprogram. This is done by executing the command **lcode** when the simulator prompts MSIM>. The simulator then asks for the object microprogram and the user should provide object file created by meta-assembler.

The data memory can be loaded by executing the command **ldm.**
For example:
To load the simulator program memory with microcode, the command is

```
MSIM>lcode <cr>
-file:mic1.obj <cr>
```

To load the data memory with integer type values, the command is

```
MSIM>ldm <cr>
-Address:0 <cr>
length  :5 <cr>
   1
   f2h
   4o
   1011b
   200 <cr>
```

Thus a binary, octal, decimal and hexa-decimal inputs can be given.

3.4.2 *Interrupts*

The simulator has internal cycle counter to model interrupting devices. The user can activate any one of the eight interrupts, specifying the number and period between two successive interrupts. After this period, the program counter value of sequencer is replaced with the corresponding interrupt vector. At startup of the simulator, all interrupts are disabled. To enable the interrupt, **setint** command is given as shown below :

```
MSIM>setint <cr>
-number:2 <cr>
```

-period (in cycles): 25 <cr>

To disable the interrupt, the period should be zero.

3.4.3 *Running the Simulator*

After supplying the simulator with all necessary requirements, such as loading of program memory and data memory, and setting of interrupts, it is time to execute the microprogram.

The simulator takes the microinstruction in hex code and changes it into binary bits and places the microinstruction in an array of characters. As explained earlier, each instruction consists of a set of fields. Using the structure of each field, the simulator separates each field and then executes the operation of the respective field's function. The various steps followed by the simulator when **run** command is given are shown in Fig 3.4a,b,c & d (for the system with one Arithmetic unit and one RAM).

The simulator can also execute one microinstruction at a time, and halts after completion of that microinstruction (i.e. single stepping). to do this, the command to be given is:

MSIM> ss <cr>

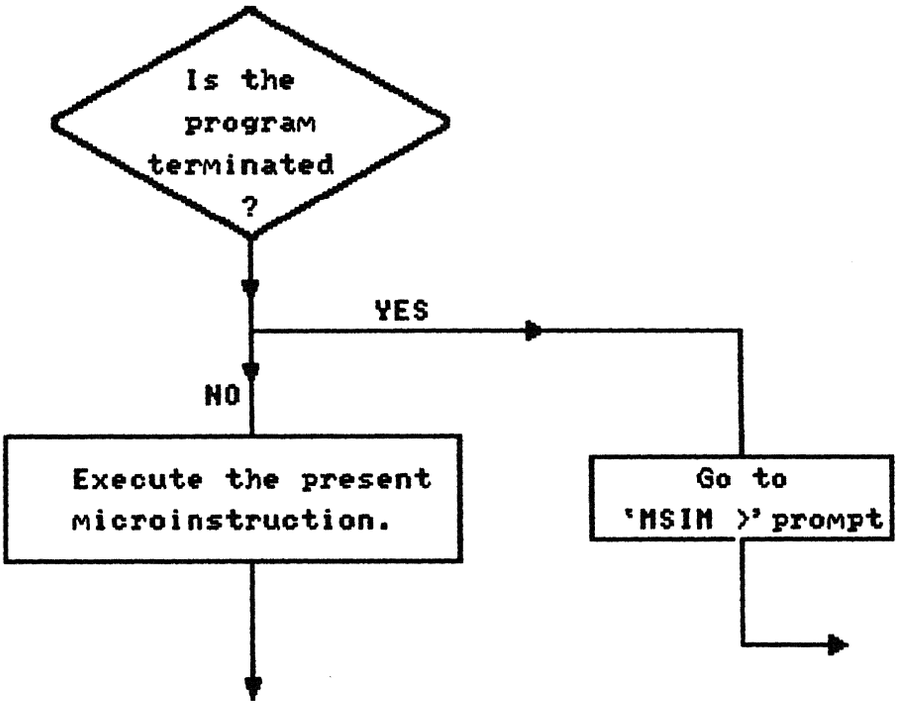


Fig 3.4.a Execution of 'run' command

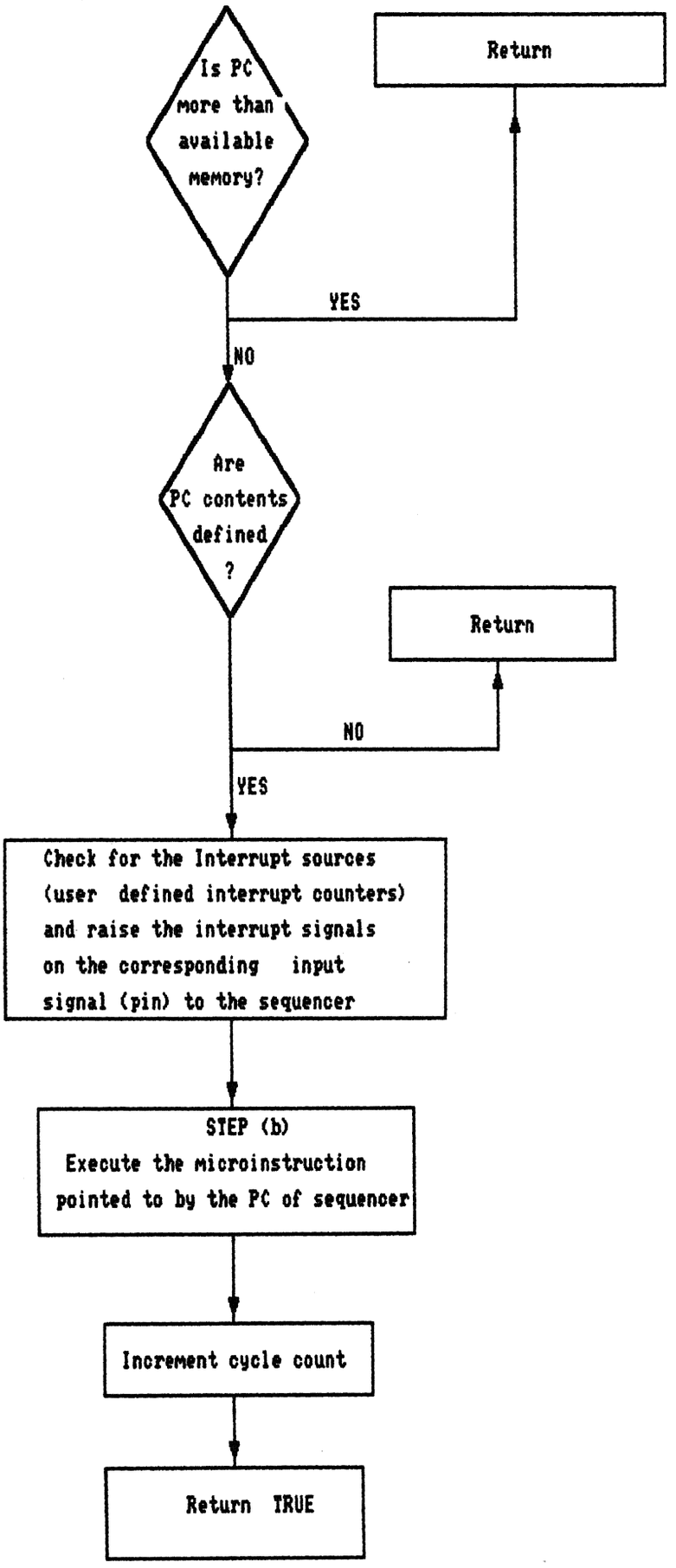


Fig 3.4.b Execution of microinstruction

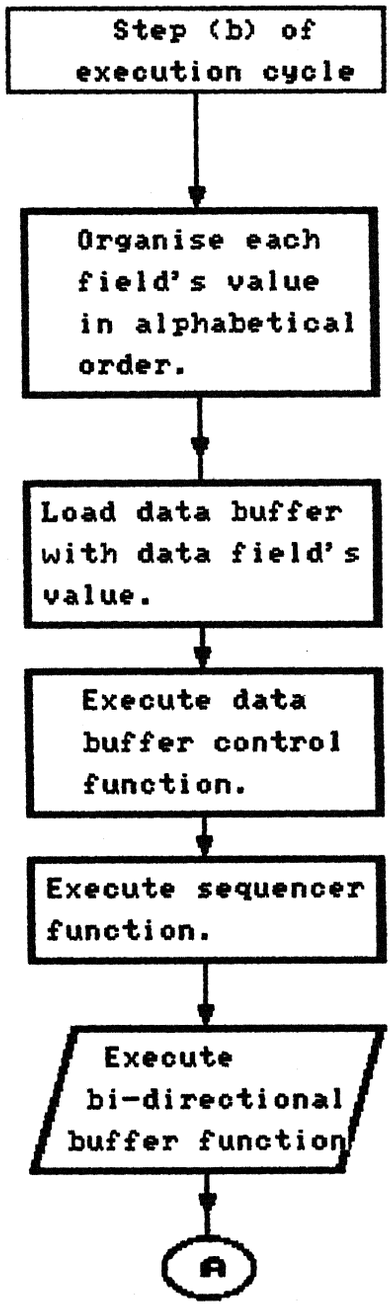


Fig. 3.4.c Execution of Step (b)

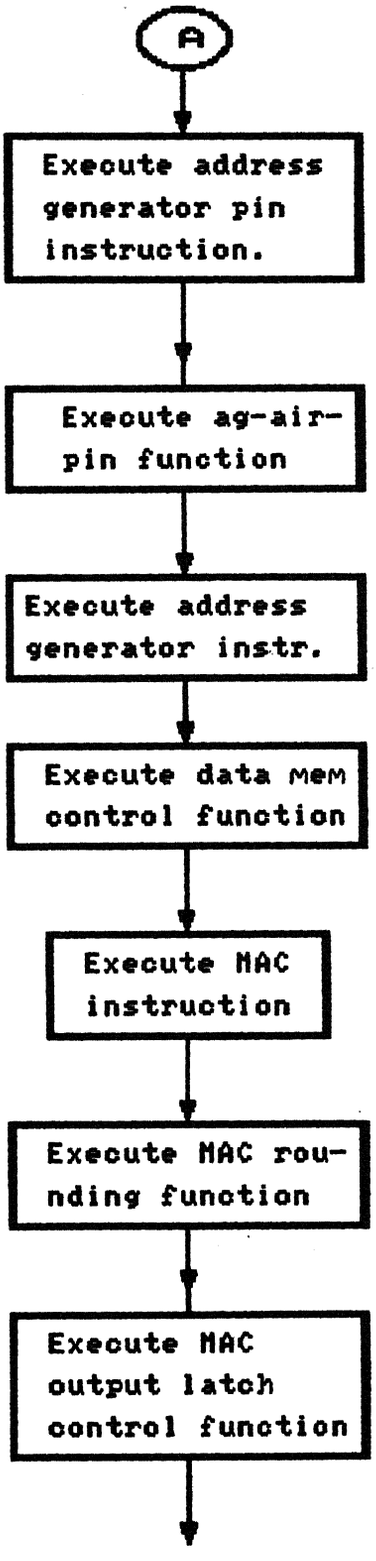


Fig 3.4.d Continuation of Step(b)

3.4.4 *Displaying Reg/Mem contents*

The simulator can display

- data memory
- program memory
- sequencer's reg/mem components contents
- address generator reg contents
- Arithmetic unit contents

The various display commands are:

1.MSIM>ddm <cr> ; 10 locations of datamemory are displayed on the screen.

-Address:

2.MSIM>dpm <cr> ; All the defined program memory contents starting from the given address are displayed on the screen.

-Address:

3.MSIM>d_seq <cr> ; displays the contents of all the counters, RAM locations, program counter, stack pointers.

4. MSIM>d_agr <cr> ; displays the contents of all the registers of the address generator.

5. MSIM>d_mac <cr> ; displays Arithmetic unit's contents.

6. MSIM>dispint <cr> ; displays interrupt vector number and period.

The data memory contents of the simulator can be dumped into a file for further analysis with the command:

```
MSIM>dumpdm <cr>
-address:  —— <cr>
-length:   —— <cr>
-file:     —— <cr>
```

3.4.5 *Ending a Simulator Session*

User can come out of a simulator session with the command:

```
MSIM>exit <cr>
Are you sure: y or Y <cr>
```

The user can also come out of the simulator with control C option.

3.5 Simulator Commands Summary

Loading the simulator:

ldm - loads data memory

ld1m - loads second data memory

(This is only for the system with 2 memory components).

lcode -loads the program memory with microprogram object file.

Interrupts:

setint -sets interrupt labels to sequencer.

dispint-displays the interrupt vectors.

Running of simulator:

run - runs the microprogram until it completes.

ss - runs single microinstruction and halts.

setpc - sets the program counter value to required location.

Display:

ddm - displays data memory contents.

ddlm - displays second data memory contents. (This is only for the system with 2 memory Components).

dpm - displays all the defined prgram memory contents starting from the given address

d_seq - displays sequencer's contents.

d_agr - displays address generator contents

d_mac - displays Arithmetic unit contents.

dumpdm -dumps data memory contents into a file.

dumpldm-dumps second data memory contents into a file.

cycle - displays a cycle count of simulator.

help - displays various command listings of the simulator

exit - exits from the simulator session .

CHAPTER 4

TESTING THE SIMULATOR

In this chapter microprograms for two algorithms,namely, 1-D convolution and matrix multiplication ,are developed to test the three proposed microcoded systems using the meta-assembler (described in Section 4.4),which converts microprograms to machine code.

4.1 System 1: One memory unit and one MAC

4.1.1 *Algorithm 1 :Matrix Multiplication*

Consider C = AB , where A is M×N and B is N×K so that C is M×K. . For purpose of illustration , let A be (4×4) and B be (4×6).The matrix $A = [a_{m,n}]$ is stored in Data-memory row-wise while matrix $B = [b_{n,k}]$ is stored in same memory column-wise [Fig 4.1].

The microprogram shown in Fig 4.2 performs the following computation :

DO 1 for m = 0 to (M-1)

DO 2 for k = 0 to (K-1)

SUM = 0

DO 3 for n = 0 to (N-1)

$c_{m,k} = SUM + a_{m,n} b_{n,k}$

3 CONTINUE

2 CONTINUE

1 CONTINUE

Table 4.1 shows the microoperations on a cycle-by-cycle basis. The major steps can be summarised as follows:

i) Cycle 1-10 involves,

- initialisation of pointers to memory locations corresponding to $a_{m,n}$, $b_{n,k}$ & $c_{m,k}$ respectively.
- setting loop counters for n,k,m.
- Other bookkeeping operations.

ii) Actual computation starts in cycle 11.

cycle

11 (1). Load a_{oo} to X-reg

12 (2) Load b_{oo} to Y-reg; (X) ⟶ (XD);

Initiate MAC operation [Note operands have to be fed in successive cycles because ADSP 1110A has a single I/O port.]

14 (4) Result ($a_{oo}b_{oo}$+0) is clocked into MR-reg and MAC operation continues.

20 (10) Load zero to Y-reg (SUM = 0);

result c_{oo} ⟶ (MR-reg) [initialising for next c_{mk}].

22 (12) (MR) = c_{oo} is output to BUS and written into memory

[Computation of each c_{mk} effectively takes 12 cycles]

iii) Initialising for the next element c_{mk} ;

a_{oo}
a_{o1}
a_{o2}
a_{o3}
a_{10}
a_{11}
⋮
a_{32}
a_{33}
b_{oo}
b_{10}
b_{20}
b_{30}
b_{o1}
b_{11}
⋮
b_{35}
c_{oo}
c_{o1}
⋮
c_{35}

Fig 4.1

D-mem contents

Table 4.1

MATRIX Vs MATRIX

$[C_{M\times K}] = [A_{M\times N}] \times [B_{N\times K}]$

M = 4 N = 4 K = 6

cycles	
	Initialisation :
1	Data = 50h ⟶ (b0))(AG) ;initialise (bo)-reg
2	Data = 100h ⟶ (i1)(AG) ;initialise i_1-reg
3	Data = 150h ⟶ (r2)(AG) ;initialise r_2-reg which as pointer to $c_{m,k}$
4	Data = N ⟶ (b1)(AG) ⟶ ; load b_1- reg with offset = N
5	Data = 0 ⟶ (LS)(MAC) ;initialise LS-reg of MAC
6	Data = 1 ⟶ LSP of SEQ ;initialise LSP to 1 .
7	Data = label4 ⟶ RAM (1) (SEQ) ;push label4 to stack .
8	Data = M-2⟶ (c2)(SEQ) ;set cntr c_2 to M-2 for looping M times
label5 9	Data = K-2 ⟶ (c1)(SEQ) & (i1)(AG) ⟶(r1)(AG) ;set cntr c_1 to K-2 for lopping K times & (i1) ⟶ (r1),r1 acts as pointer to b_{nk}
label3 10	Data =N-2 ⟶ (c0)(SEQ) & (b0)(AG) ⟶(r0)(AG) ;set cntr c_0 to N-2 for looping N times & (b0) ⟶ (r0),r0 acts as pointer to a_{mn}
label2 11	COMPUTATION starts

	X	XD	Y	MR	I/O	MEM
11(1)	a_{00}					
12(2)		a_{00}	b_{00}			
13(3)	a_{01}					
14(4)		a_{01}	b_{10}	$\Sigma=a_{00}b_{00}$		
15(5)	a_{02}					
16(6)		a_{02}	b_{20}	$\Sigma=\Sigma+a_{01}b_{10}$		
17(7)	a_{03}					
18(8)		a_{03}	b_{30}	$\Sigma=\Sigma+a_{02}b_{20}$		
label4 19(9)	Continues to next sequential microprogram location					
20(10)			0	c_{00}		
21(11)	Continues to next sequential microprogram location					
22(12)					c_{00}	wr

23	Jumps uncondiionally to label3 for K times
93	Decrements counter c2 and (b0)(AG) ⟶ (r5)(AG)
94	(b1)(AG) is added to (r5)(AG) & checks c2 for completion of all rows
95	Repeats the whole operation for M times by jumping unconditionally to label5 & (r5)(AG) ⟶(b0)(AG)

```
        p_org                                   100h
        d_org                                   50h
              M                 equ             4
              N                 equ             4
              K                 equ             6
2 50h & enable & yrtb(b0)(r0) & dsel
2 100h & enable & dti(il)
2 150h & enable & btr(b3)(r2)
2 N & enable & yrtb(bl)(r0) & dsel
enable & rdlatch & bus
2 1 & enable & wrrsp
2 label4 & enable & psdss
2 M-2 & enable & wrcntr(c2)
label5: wrcntr(cl) & enable & 2 K-2 & itr(il)(rl)
label3: wrcntr(c0) & enable & 2 N-2 & btr(b0)(r0)
label2: yinc(c0)(r0) & rd & x=bus
yinc(cl)(rl) & rd & y=bus_ckmr_xus*yus+mr
& branch(unconditional)(c0) & 2 label2 & enable
label4: cont
enable & y=bus_ckmr_xus*yus & dccntr(cl) & rdlatch
cont
bus=ls & macenable & yinc(c2)(r2) & wr & jtwo(sign)(cl)
jda(unconditional) & 2 label3 & enable
dccntr(c2) & btr(b0)(r5)
jtwo(sign)(c2) & yadd(c0)(bl)(r5)
jda(unconditional) & 2 label5 & enable & yrtb(b0)(r5)
```

Fig 4.2 Microprogram Matrix multiplication for System 1

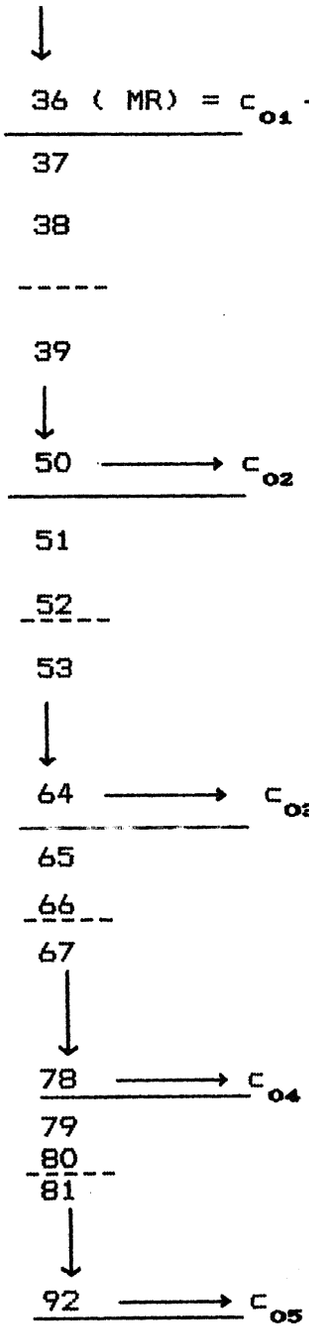
23 Sequencer generates next microinstruction address = label3

24 Microoperation in label3 are performed :

- set up loop counter for N :(N-2) ⟶ (counter co)
- initialise pointer to start of next column b_{nk}.

iv) Computation of next c_{mk} i.e c_{o1} starts .

25 Similar to cycles 10-21.



v) After completion of all K elements c_{mk}, k = 0,....(K-1) of a row (corresponding to jump unconditionally to label3 i.e cycle 22 of Table 4.1), computation of next row $c_{m+1,k}$ starts .

93 Counter c_2 (loop counter for m) is decremented & (b0) = 50h ⟶ (r5)

94 check counter c_2 for completion of all rows &(b1)=N+(r5) ⟶ (r :Start address of next row is generated by adding offset = N.

95 If m < M jumps to label5 & (r5) ⟶ b_o : b_o now points to start of next row.

Results & discussion

1. i) For a (4x4) x (4x6) multiplication, each c_{ij} element requires 4 MAC operations resulting in a total of (4 MAC/ element) x 24 = 96 MAC operations.

ii) A total of 355 clock cycles are required for getting the product matrix C . Assuming a clock cycle of 100 nsec [ADSP 1110A can operate with a maximum clock period of 50 nsec]

$$\text{Throughput achieved} = \frac{96}{100 \times 10^{-9} \times 355} = 2.71 \text{ MOPS}$$

2. i) For a [10x10] x [10x20] multiplication ,

Total No.of MAC operations = (10 MAC/element) x 200 elements = 2000

ii) Total No. of cycles required = 5237.

$$\text{Throughput achieved} = \frac{2000}{10^{-9} \times 5237} = 3.82 \text{ MOPS}$$

4.1.2 *Algorithm 2 : 1-D Convolution*

The convolution of two sequences

$$x(n) = \{ x(0), x(1), \ldots\ldots x(M-1) \},$$

$$h(n) = \{ h(0), h(1), \ldots\ldots h(N-1) \}$$

is given by,

$$y(n) = \sum_{k=0}^{N-1} h_k x_{n-k} \quad , \; n = 0, 1, \ldots M+N-1.$$

Let,

$$x(n) = 0 \text{ , for } n < 0,$$

$$h(n) = 0, \text{ for } n < 0.$$

h(n) and x(n) elements are stored in data memory starting at location 50h and 100h respectively.The microprogram for this algorithm is shown in Fig 4.3, and the corresponding cycle-by-cycle operations are explained in Table 4.2.

The microprogram shown in Fig 4.3 performs the following operations,

(i) Initialisation :

Cycle 1-8 involves,

- Setting loop counters for M+N-1 and N
- initialising the LS-reg of MAC to zero
- other book-keeping operations.

(ii) Computation :

Actual computation starts in cycle 9,

Table 4.2
1-D Convolution
$y(k) = x(m) * h(n)$.

$M = 16$; $N = 5$; $K = M + N - 1 = 20$

ycles	
	Initialisation
1	Data = 50h ⟶ (i0)(AG) ; initialise (i0) register
2	Data = 100h ⟶ (r1)(AG) ; initialise r1, which acts as pointer to x(n).
3	Data = 150h ⟶ (r2)(AG) ; initialise r2, which acts as pointer to y(k).
4	Data = 0 ⟶ (LS)(MAC) ; initialise LS-reg of MAC
5	Data = 1 ⟶ (LSP)(SEQ) ; initiate LSP to 1.
6	Data = label3 ⟶ RAM(1)(SEQ) ; push label3 to STACK
7	Data = M + N - 3 ⟶ (c1)(SEQ) ; set counter c_1 to M+N-3 for looping M+N-1 times.
abel1 8	Data = N - 2 ⟶ (c0)(SEQ) & (i0)(AG) ⟶ (r0)(AG) ; set counter c_0 to N-2 for looping N times
abel2	Computation

	X	XD	Y		I/O	MEM
9(1)	h_0					
0(2)		h_0	x_0			
11(3)	h_1					
12(4)		h_1	0	$\Sigma = h_0x_0$		
13(5)	h_2					
14(6)		h_2	0	Σ		
15(7)	h_3					
16(8)		h_3	0	Σ		
17(9)	h_4					
18(10)		h_4	0	Σ		

19(11)	Data = N+1 is added to (r1)(AG) & Decrements c1

20(12)			0	y_0		

21(13)	Continues to next sequential microprogram location.

22(14)					y_0	wr

23	Repeats the computation for M+N-1 times by jumping to label1

9(1) loads h_o to X-reg.

10(2) loads x_o to Y-reg; X ⟶ XD; initiation of MAC operation

↓

12(4) Result ($a_o h_o$+0) is clocked into MR-reg and MAC

↓

19(11) Data = N + 1 is added to (r1), which then pointes to x_1 for computation of next element , y_1.

20(12) Load zero to Y-reg ; result y_o ⟶ MR-reg.

[initialisation for the next y_n]

22(14) MR = y_o is output to BUS and written into memory

[Computation of each y_n effctively takes 14 cycles].

(iii) Initialisation for next element (y_n) :

23 Sequencer generates NEXT microinstruction address = label1.

24 Microoperations in label1 are performed ,i.e,

- Set up loop counter for N : (N-2) ⟶ (c0)
- Initiate pointer to starting location of coefficient elements h(n).

(iv) Computation of next element y_1 srarts,

25 (Similar to 8-21)

↓

38 ⟶ (MR) = y_1 ⟶ BUS ⟶ Memory.

39

40

41

↓

54 ⟶ y_2

⋮

326 ⟶ y_{20}

9(1) loads h_o to X-reg.

10(2) loads x_o to Y-reg; X ⟶ XD; initiation of MAC operation

↓

12(4) Result ($a_o h_o$+0) is clocked into MR-reg and MAC

↓

19(11) Data = N + 1 is added to (r1), which then pointes to x_1 for computation of next element , y_1.

20(12) Load zero to Y-reg ; result y_o ⟶ MR-reg.

[initialisation for the next y_n]

22(14) MR = y_o is output to BUS and written into memory

[Computation of each y_n effctively takes 14 cycles].

(iii) Initialisation for next element (y_n) :

23 Sequencer generates NEXT microinstruction address = label1.

24 Microoperations in label1 are performed , i.e,

- Set up loop counter for N : (N-2) ⟶ (c0)
- Initiate pointer to starting location of coefficient elements h(n).

(iv) Computation of next element y_1 srarts,

25 (Similar to 8-21)

↓

38 ⟶ (MR) = y_1 ⟶ BUS ⟶ Memory.

39

40

41

↓

54 ⟶ y_2

⋮

326 ⟶ y_{20}

```
p_org           50h
d_org           50h

N       equ     5
M       equ     16

2 50h & enable & dti(i0)
2 100h & enable & btr(b3)(r2)
2 150h & enable & btr(b3)(r2)
 enable & rdlatch & ls=bus
2 1 & enable & wrrsp
2 label3 & enable & psdss
wrcntr(c1) & 2 M+N-3 & enable
label1: wrcntr(c0) & 2 N-2 & enable & itr(i0)(r0
label2: yinc(c0)(r0) & rd & x=bus
branch(unconditional)(c0) & ydec(c1)(r1) & rd & 2 label2
& enable & y=bus_ckmr_xus*yus+mr
label3: 2 N+1 & enable & yadd(c0)(b3)(r1) & dccntr(c1)
enable & y=bus_ckmr_xus*yus & rdlatch
cont
bus=ls & macenable & jtwo(sign)(c1) & yinc(c2)(r2) & wr
jda(unconditional) & 2 label1 & enable
```

Fig 4.3 : Microprogram for 1-D convolution for System 1

Results & Discussion

1 (i) For a [5x16] convolution, each element y_n requires 5 MAC operations resulting a total of [5MAC/element] x 20 = 100 MAC operations.

(ii) A total of 326 clock cycles are required for getting the convolved sequence, y_n.

$$\text{Throughput} = \frac{100}{100 \times 10^{-9} \times 326} = 3.08 \text{ MOPs.}$$

2 (i) For a [10x30] convolution,

Total MAC operations = 390.

(ii) Total number of clock cycles = 1020.

$$\text{Throughput achieved} = \frac{390}{100 \times 10^{-9} \times 1020} = 3.82 \text{ MOPs}$$

4.2 System 2: One memory-Two MAC

4.2.1 ***Algorithm 1 : Matrix Multiplication***

In this architecture, there are two processors and one shared memoryunit connected via a common bus. In order to effectively

utilise the two processors two elements $c_{m,k}$ and $c_{m,k+1}$ of a row are calculated simultaniously .

For example ,

$$c_{oo} = a_{oo}b_{oo} + a_{o1}b_{1o} + a_{o2}b_{2o} + a_{o3}b_{3o} .$$

$$c_{o1} = a_{oo}b_{o1} + a_{o1}b_{11} + a_{o2}b_{21} + a_{o3}b_{31} .$$

p_org	50h
d_org	50h
M	4
N	4
K	6

```
2 50h & enable & yrtb(b0)(r0) & dsel
2 100h & enable & dti(i1)
2 100h+N & enable & dti(i2)
2 150h & enable & btr(b3)(r3)
enable & rdlatch & rdlatch1 & ls=bus & ls=bus1
2 1 & enable & wrrsp
2 label3 & enable & psdss
2 M-2 & enable & wrcntr(c2)
label4: 2 k/2-2 & enable & wrcntr(c1) & itr(i1)(r1)
itr(i2)(r2)
label2: 2 N-2 & enable & wrcntr(c0) & btr(b0)(r0)
label1: yinc(c0)(r0) & rd & x=bus & x=bus1 & rdlatch1
yinc(c1)(r1) & rd & y=bus_ckmr_xus*yus+mr
rdlatch1 & yinc(c2)(r2) & rd & y=bus_ckmr_xus*yus+mr1 &
```

```
branch(unconditional)(c0) & 2 label1 & enable
label3: cont
enable & y=bus_ckmr_xus*yus & y=bus_ckmr_xus*yus1
& rdlatch & rdlatch1
2 N & enable & yadd(c2)(b3)(r1)
bus=ls & macenable & yinc(c2)(r3) & wr
bus=ls1 & mac1enable & yinc(c2)(r3) & wr & dccntr(c1) & wrlatch1
jtwo(sign)(c1) & 2 N & enable & yadd(c2)(b3)r2)
jda(unconditional) & 2 label2 & enable
dccntr(c2) & btr(b0)(r5)
jtwo(sign)(c2) & 2 N & enable & yadd(c0)(b3)(r5)
jda(uncoditional) & 2 label4 & enable & yrtb(b0)(r5)
```

Fig 4.4 : Microprogram for Matrix multiplication for System 2

As shown in Table 4.3(the corresponding microprogram is shown is Fig 4.4), in cycle 1 row element a_{oo} isfed to both MACs simultaneously . In cycle 2 , column element b_{oo}(first column) is fed to MAC 1 while in cycle 3, column element b_{o1}(second column) is fed to MAC 2 . In contrast to System 1 (1MAC & 1Memory), where operands could be supplied to the MAC every second cycle , in this architecture (2 MAC & 2 Memory) operands can be suplied to the respective MACs only every third cycle due to common-bus configuration .

Table 4.3

MATRIX Vs MATRIX

$[A]_{MXN} \times [B]_{NXK} = [C]_{MXK}$

M = 4 N = 4 K = 6

	Initialisation:
1	Data = 50h ⟶ (b0)(AG)
2	Data = 100h ⟶ (i1)(AG)
3	Data = 100h+1 ⟶ (i2)(AG)
4	Data = 150h ⟶ (r3)(AG)
5	Data = 0 ⟶ (LS) of MAC1 and MAC2
6	Data = 1 ⟶ LSP (SEQ)
7	Data = label3 ⟶ RAM(1) (SEQ)
8	Data = M-2 ⟶ (c2)(SEQ)
label4	Data = K/2-2 ⟶ (c1)(SEQ) & (i1)(AG) ⟶ (r1)(AG)
10	(i2)(AG) ⟶ (r2)(AG)
label2 11	Data =N-2 ⟶ (c0)(SEQ) & (b0)(AG) ⟶ (r0)(AG)
label1 12	Computation starts

	X	XD	Y	MR	I/O	X	XD	Y	MR	I/O	MEM
12(1)	a_{00}					a_{00}					
13(2)		a_{00}	b_{00}								
14(3)							a_{00}	b_{01}			
15(4)	a_{01}					a_{01}					
16(5)		a_{01}	b_{10}	$\Sigma=a_{00}b_{00}$							
17(6)							a_{01}	b_{11}	$\Sigma=a_{00}b_{01}$		
18(7)	a_{02}					a_{02}					
19(8)		a_{02}	b_{20}	$\Sigma=\Sigma+a_{01}b_{10}$							
20(9)							a_{02}	b_{21}	$\Sigma=\Sigma+a_{01}b_{11}$		
21(10	a_{03}					a_{03}					
22(11		a_{03}	b_{30}	$\Sigma=\Sigma+a_{02}b_{20}$							
23(12							a_{03}	b_{31}	$\Sigma=\Sigma+a_{02}b_{21}$		
24(13	Continues to next microprogram location.										
25(14			0	c_{00}				0	c_{01}		

26(15)	Data = N is added to (r1)(AG)												
27(16				C_{oo}								wr	
28(17											C_{o1}	wr	
29(18	Data = N is added to (r2)(AG).												
30	Repeats the above computation for K/2 times by jumping uncondition- to label2.												
⋮													
l	Decrements counter (c2)(SEQ) & (b0)(AG) ⟶ (r5)(AG).												
l+1	Data = N is added to (r5)(AG).												
l+2	Repeats the above sequence for M times by jumping unconditionally												
	to label4 & (r5)(AG) ⟶ (b0)(AG).												

It can be seen from Table 4.3 that on the average 4 (5) MAC operations are performed in 7 (8) cycles , whereas in System 1 , the rate is 3 (4) MACs per 7(8) cycles .Thus with 2MACs , the speedup is by a factor of 1.25 to 1.33 .

Results & Discussion

1. [4x4] x [4x4] multiplication ⟶ 263 cycles

$$\text{Throughput} = \frac{96}{10^{-7} \times 263} = 3.66 \text{ MOPs}$$

2. [10x10] x [10x20] multiplication ⟶ 3847 cycles

$$\text{Throughput} = \frac{2000}{10^{-7} \times 3847} = 5.2 \text{ MOPs}$$

Comparing with the results of System 1 , we concluded that Systems 2 attain a speed-up factor of about 1.35 – 1.36 . Theoritical speed-up factor of 2 cannot be approached because of the single-bus configuration .

4.2.2 *Algorithm 2 : 1-D convolution*

In this the x-sequence is made two parts and stored in data memory at separate locations FBh and 125h respectively.The overlapping

terms are calculated first. While calculating non-overlapping terms, the coefficient elements are fed to both MACs simultaneously.

The microprogram for this algorithm is shown in Fig 4.5 and explained on a cycle-by-cycle basis in Table 4.4.

```
p_org                    50h

N          equ           5
M1         equ           8

2 50h & enable & dti(i0)
2 50h+N-1 & enable & yrtb(b0)(r0) & dsel
2 100h-N+2 & enable & btr(b3)(r2)
2 125h & enable & btr(b3)(4)
2 200h & enable & btr(b3)(r6)
2 1FFh & enable & btr(b3)(r7)
enable & rdlatch & rdlatch1 & ls=bus & ls=bus1
2 1 & enable & wrrsp
2 label3 & enable & psdss
2 label8 & enable & psdss
2 M1-2 & enable & wrcntr(c3)
2 N-3 & enable & wrcntr(c2) & btr(b0)(r5)
label6: 2 M-3 & enable & wrcntr(c0) & itr(i0)(r0)
label1: yinc(c0)(r0) & rd & x=bus
ydec(c0)(r4) & rd & y=bus_ckmr_xus*yus+mr
yinc(c1)(r2) & rd x=bus1 & rdlatch1
```

```
ydec(c0)(r5) & rd & y=bus_ckmr_xus*yus+mr1 & rdlatch1 &
branch(unconditional) & 2 label1 & enable
label3: dccntr(c2) & 2 N & enable & yadd(c0)(b3)(r4)
2 1 & enable & x=bus1 & rdlatch1 & ckmr & rdlatch
bus=ls & macenable & y=bus_ckmr_xus*yus+mr1 & rdlatch1
cont
enable & y=bus_ckmr_xus*yus & y=bus_ckmr_xus*yus1 & rdlatch &
rdlatch1 & btr(b0)(r5)
2 N-2 & enable & ysub(c0)(b3)(r2)
bus=ls1 & mac1enable & yinc(c3)(r6) & wr & wrlatch1 &
jtwo(sign)(c2)
jda(unconditional) & 2 label6 & enable
ydec(c0)(r2)
label9: 2 N-2 & enable & wrcntr(c1) & itr(i0)(r0)
label7: yinc(c0)(r0) & rd & x=bus & x=bus1 & rdltch1
ydec(c0)(r4) & rd & y=bus_ckmr_xus*yus+mr
ydec(c0)(r2) & rd & y=bus_ckmr_xus*yus+mr1 & rdlatch1 &
branch(unconditional)(c1) & 2 label7 & enable
label8: 2 N+1 & enable & yadd(c0)(b3)(r4) & dccntr(c3)
enable & y=bus_ckmr_xus*yus & y=bus_ckmr_xus*yus1 & rdlatch
& rdlatch1
2 N-1 & enable & yadd(c0)(b3)(r2)
bus=ls & macenable & yinc(c0)(r6) & wr
bus=ls1 & mac1enable & ydec(c0)(r7) & wr & jtwo(sign)(c3)
& wrlatch1
jda(unconditional) & 2 label9 & enable
```

Fig 4.5 : Microprogram for 1-D convolution for System 2

Table 4.4
1-D Convolution

$y(k) = x(m) * h(n).$

$M_1 = 8$; $M_2 = 8$; $N = 5$; $K = M_1 + M_2 + N - 1$

	Initilisation
1	Data = 50h ⟶ (i0)(AG)
2	Data = 50h+N-1 ⟶ (b0)(AG)
3	Data = 100h-N+2 ⟶ (r2)(AG)
4	Data = 125h ⟶ (r4)(AG)
5	Data = 200h ⟶ (r6)(AG)
6	Data = 1FFh ⟶ (r7)(AG)
7	Data = 0 ⟶ (LS-reg)(MAC1 and MAC2)
8	Data = 1 ⟶ (LSP)(SEQ)
9	Data = label3 ⟶ RAM(1)(SEQ)
10	Data = label8 ⟶ RAM(2)(SEQ)
11	Data = $M_1 - 2$ ⟶ (c3)(SEQ)
12	Data = $N^1 - 3$ ⟶ (c2)(SEQ) & (b0)(AG) ⟶ (r5)(AG)
label6 13	Data = N - 3 ⟶ (c0)(SEQ) & (i0)(AG) ⟶ (r0)(AG)

label1	Computation	MAC1				MAC2					
	X	XD	Y	MR	I/O	X	XD	Y	MR	I/O	MEM
14(1)	h_0										
15(2)		h_0	x_0								
16(3)						x'_4					
17(4)							x'_4	h_4			
18(5)	h_1										
19(6)		h_1	0	$\Sigma = h_0 x_0$							
20(7)						x'_5					
21(8)							x'_5	h_3	$\Sigma=\Sigma+x'_4 h'_4$		
22(9)	h_2										
23(10)		h_2	0	Σ							
24(11)						x'_6					
25(12)							x'_6	h_2	$\Sigma = \Sigma + x'_5 h_3$		
26(13)	h_3										
27(14)		h_3	0	Σ							

28(15)						x'_7						
29(16)							x'_7	h_1	$\Sigma=\Sigma + x'_6 h_2$			
30(17)	Data = M_1 is added to (r4)(AG) & Decrements (c2)(SEQ).											
31(18)				Σ		1						
32(19)					Σ		1	Σ	$\Sigma=\Sigma + \Sigma*1$			
33(20)	Continues to next microprogram location.											
34(21)			0					0	y_8			
35(22)	Data = M_1 - 2 is substracted from (r2)(AG)											
36(23)										y_8	wr	
37(24)	Repeats the above operation for N - 1 times by jumping unconditionally to label6.											
110	Decrements (r2)(AG).											
label19 111	Data = N-2 ⟶ (c1)(AG) & (i0)(AG) ⟶ (r0)(AG)											
label17 112(1)	h_0					h_0						
113(2)		h_0	x_4									
114(3)							h_0	x'_7				
115(4)	h_1					h_1						
116(5)		h_1	x_3	$\Sigma=h_0 x_4$								
117(6)							h_1	x'_6	$\Sigma=\Sigma+h_0 x'_7$			
118(7)	h_2					h_2						
119(8)		h_2	x_2	$\Sigma=\Sigma+h_1 x_3$								
120(9)							h_2	x'_5	$\Sigma=\Sigma+h_1 x'_6$			
121(10	h_3					h_3						
122(11)		h_3	x_1	$\Sigma=\Sigma+h_2 x_2$								
123(12)							h_3	x'_4	$\Sigma=\Sigma+h_2 x'_5$			
124(13	h_4					h_4						

125(14		h_4	x_0	$\Sigma=\Sigma+h_3x_1$								
126(15							h_4	x'_3	$\Sigma=\Sigma+h_3x'_4$			
127(16	Data=N+1 is added to (r4)(AG) & Decrements counter (c3)(SEQ)											
128(17			0	$\Sigma=\Sigma+h_4x_0$				0	$\Sigma=\Sigma+h_4x'_3$			
129(18	Data = N-1 is added to (r2)(AG).											
130(19)					y_{12}							wr
131(20										y_7		wr
132	Repeats the above computation for M_1 times by jumping unconditionally to label9.											

Results & Discussion

1 (i) [5×16] convolution ⟶ 287 cycles.

(ii) $$\text{Throughput} = \frac{100}{10 \times 10^{-9} \times 287} = 3.5 \text{ MOPs.}$$

2 (i) [10×30] convolution ⟶ 971 cycles.

(ii) $$\text{Throughput} = \frac{390}{10 \times 10^{-9} \times 971} = 4.02 \text{ MOPs.}$$

Comparing to System 1, we conclude that System 2 attains a speed-up factor of about 1.2. Theoritical speed-up factor of 2 cannot be achieved because of the single bus configuration.

4.3 System 3 : 2 MACs and 2 memory units

In this architecture , each MAC has its own local memory connected through local bus .The two units are then connected to the HOST by a common bus .

4.3.1 *Algorithm 1 : Matrix Multiplication*

In this architecture , B matrix is stored in both the data memories . Half of A matrix is stored in one data memory (Mem 1) and the other half is stored in the other data memory (Mem 2) Fig 4.6 .

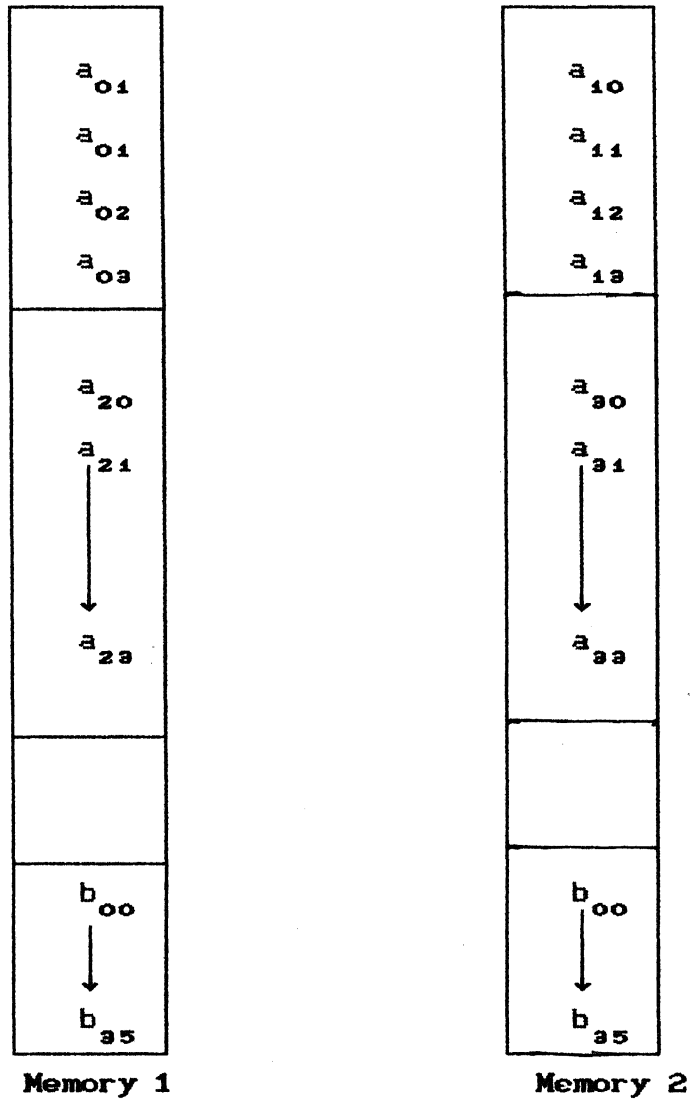


Fig 4.6 Storing of the matrices in two separate memories

As shown in Table 4.5(the corresponding microprogram is given in Fig 4.7) two separate row elements $c_{m,k}$(i.e c_{oo}) and $c_{m+1,k}$ (i.e c_{10}) are calculated simultaneously in MAC1 and MAC2 respectively . The microoperations shown in Table 4.1 for 1MAC-1MEM case are essentially duplicated and executed in parallel in the two MACs thereby doubling the throughput . MAC1 outputs c_{oo} in the 12^{th} "computation cycle"as in Table 1 and written into Memory 1. The element c_{10} which is available in MR-reg (MAC2) at the same time as c_{oo}, can however be output on the bus only after c_{oo}, i.e in 13^{th} computation cycle due to common bus-structure . Here we are assuming that product matrix is stored in one memory row-wise .

```
p_org                    100h
M          equ           4
N          equ           4
K          equ           6
```

```
2 50h & enable & yrtb(b0)(r0) & dsel & yrtb1(b0)(r0) & dsel
2 100h & enable & dti(i1) & dti1(i1)
2 150h & enable & btr1(b3)(r2)
2 150h+K & enable & btr1(b3)(r3)
enable & rdlatch & rdlatch1 & ls=bus & ls=bus1
2 1 & enable & wrrsp
2 label3 & enable & psdss
```

```
2 label2 & enable & psdss
2 M/2-2 & enable & wrcntr(c2)
label5: 2 K-2 & enable & wrcntr(c1) & itr(i1)(r1) & itr1(i1)(r1)
label4: 2 N-2 & enable 7 wrcntr(c0) & btr(b0)(r0) & btr1(b0)(r0)
label1: yinc(c0)(r0) & rd & x=bus & yinc1(c0)(r0) & rd1 & x=bus1
yinc(c0)(r1) & rd & y=bus_ckmr_xus*yus+mr & yinc1(c0)(r1) & rd1
& y=bus_ckmr_xus*yus+mr1 & 2 label1 & enable &
branch(unconditional)(c0)
label3: cont
enable & rdlatch & rdlatch1 & y=bus_ckmr_xus*yus &
y=bus_ckmr_xus*yus
cont
wrlatch & bus=ls & macenable & rdlatch & wr1 7 yinc1(c1)(r2)
bus=ls1 & mac1enable & yinc1(c1)(r3) & wr1
branch(unconditional)(c1) & 2 label4 & enable
label2: 2 K & enable & yadd1(c0)(b3)(r3)
yrtb(b0)(r0) & yrtb1(b0)(r0) & branch(unconditional)(c2)
2 label5 & enable
```

Fig 4.7 Microprogram for Matrix multiplication for System 3

Table 4.5

MATRIX Vs MATRIX

$$[A]_{M\times N}\ [B]_{N\times K} = [C]_{M\times K}$$

M = 4 N = 4 K = 6

	Initialisation											
1	Data = 50h ⟶ (b0)(AG1) & (b0)(AG2)											
2	Data =100h ⟶ (i1)(AG1) & (i1)(AG2)											
3	Data = 150h ⟶ (r2)(AG2)											
4	Data = 150h + k ⟶ (r3)(AG2)											
5	Data = 0 ⟶ (LS) of MAC1 and MAC2											
6	Data = label3 ⟶ RAM(1)(SEQ)											
7	Data = label2 ⟶ RAM(2)(SEQ)											
8	Data = 1 ⟶ LSP(SEQ)											
9	Data = M/2-2 ⟶ (c2)(SEQ)											
label5 10	Data=K-2⟶(c1)(SEQ) & (i1)(AG1)⟶(r1)(AG1) & (i1)(AG2)⟶(r1)(AG2)											
label4 11	Data=N-2⟶(c0)(SEQ) & (b0)(AG1)⟶(r0)(AG1) & (b0)(AG2) ⟶(r0)(AG2											
label1	Computation	MAC1					MAC2					
	X	XD	Y	MR	I/O		X	XD	Y	MR	I/O	MEM
12(1)	a_{00}						a_{10}					
13(2)		a_{00}	b_{00}					a_{10}	b_{01}			
14(3)	a_{01}						a_{11}					
15(4)		a_{01}	b_{10}	$\Sigma = a_{00}b_{00}$				a_{11}	b_{11}	$\Sigma = a_{10}b_{01}$		
16(5)	a_{02}						a_{12}					
17(6)		a_{02}	b_{20}	$\Sigma=\Sigma+a_{01}b_{10}$				a_{12}	b_{21}	$\Sigma=\Sigma+a_{11}b_{11}$		
18(7)	a_{03}						a_{13}					
19(8)		a_{03}	b_{30}	$\Sigma=\Sigma+a_{02}b_{20}$				a_{13}	b_{31}	$\Sigma=\Sigma+a_{12}b_{21}$		
20(9)	Continues to next sequential microprogramme location.											
21(10)			0	c_{00}					0	c_{10}		
22(11)	Continues to next sequential micro programme location											
23(12)					c_{00}							wr
24(13)											c_{10}	wr
1	Data = K is added to (r2)(AG2)											
1+1	Data = K is added to (r3)(AG2)											
1+2	(r0)(AG1) ⟶ (b0)(AG1) & (r0)(AG2) ⟶ (b0)(AG2) & Tests for sign of c2, if negative, jumps unconditionally to label5.											

Results and Discussion

1 . [4x4] x [4x4] multiplication ⟶ 185cycles.

$$\text{Throughput} = \frac{96}{10^{-7} \times 185} = 5.22 \text{ MOPs} .$$

2 . [10x10] x [10x20] multiplication ⟶ 2629 cycles .

$$\text{Throughput} = \frac{2000}{10^{-7} \times 2629} = 7.61 \text{ MOPs} .$$

The speed-up factor is 1.92-1.99 with reference to System 1 , thus approaching the theoritical factor of 2 .

4.3.2 *Algorithm : 1-D Convolution*

As explained earlier in Section 4.2.2 x-sequence is partitioned into two parts and the two parts are stored in separate memory units. Convolution is performed in the same manner as explained earlier. The microprogram for this in shown in Fig 4.8 and on a cycle by cycle basis in Table 4.6.

```
                p_org                       100h
                N           equ             5
                M1          equ             8

2 50h & enable & dti(i0) & dti1(i0)
2 50h+N-1 & enable & yrtb(b0)(r0) & dsel
2 100h-N+2 & enable & btr(b3)(r2)
2 125h & enable & btr1(b3)(r4)
2 150h & enable & btr1(b3)(r6)
2 14Fh & enable & btr1(b3)(r7)
enable & rdlatch & rdlatch1 & ls=bus & ls=bus1
2 1 & enable & wrrsp
2 label6 & enable & psdss
2 label7 & enable & psdss
2 M1-2 & enable & wrcntr(c3)
2 N-3 & enable & wrcntr(c2)
label2: 2 N-3 & enable & wrcntr(c0) & itr1(i0)(r0) & btr(b0)(r0)
label1: yinc(c0)(r2) & rd & x=bus & yinc1(c0)(r0) & rd1 & x=bus1
ydec(c0)(r0) & rd & y=bus_ckmr_xus*yus+mr & ydec1(c0)(r4) & rd1
& y=bus_ckmr_xus*yus+mr1 & branch(unconditional)(c0) & 2 label1
 & enable
label6: 2 N-2 & enable & rdlatch & ysub(c0)(b3)(r2)
2 1 & enable & rdlatch1 & x=bus1 & ckmr
wrlatch & bus=ls & macenable & rdlatch & y=bus_ckmr_xus*yus+mr1
2 N & enable & yadd1(c0)(b3)(r4)
```

```
enable & rdlatch & rdlatch1 & y=bus_ckmr_xus*yus & dccntr(c2)
 & y=bus_ckmr_xus*yus1
cont
bus=ls1 & mac1enable & yinc1(c2)(r6) & wr1 & jtwo(sign)(c2)
jda(unconditional) & 2 label2 & enable
ydec(c0)(r2)
label4: itr(i0)(r0) & itr1(i0)(r0) & 2 N-2 & enable & wrcnte(c1)
label3: yinc(c0)(r0) & rd & x=bus & yinc1(c0)(r0) & rd1 & x=bus1
ydec(c0)(r2) & rd & y=bus_ckmr_xus*yus+mr & ydec1(c0)(r4) & rd1
& y=bus_ckmr_xus*yus+mr1 & branch(unconditional)(c1) & 2 label3
 & enable
label7: 2 N-1 & enable & yadd(c0)(b3)(r2)
enable & rdlatch & rdlatch1 & y=bus_ckmr_xus*yus
 & y=bus_ckmr_xus*yus
2 N+1 & enable & yadd1(c0)(b3)(r4) & dccntr(c3)
wrlatch & bus=ls & macenable & rdlatch1 & wr1 & ydec1(c0)(r7)
bus=ls1 & mac1enable & wr1 & yinc1(c0)(r6) & jtwo(sign)(c3)
jda(unconditional) & enable & 2 label4
```

Fig 4.8 : Microprogram for 1-D convolution for System 3

Table 4.6

1D-CONVOLUTION

$y(k) = x(n) * h(m)$

$N = 5$; $M_1 = 8$; $M_2 = 8$; $K = N + M_1 + M_2 - 1 = 20$

	Initialisation											
1	Data = 50h ⟶ (i0)(AG1) & (i0)(AG2)											
2	Data = 50h + N - 1 ⟶ (b0)(AG1)											
3	Data = 100h - N + 2 ⟶ (r2)(AG2)											
4	Data = 125h ⟶ (r4)(AG2)											
5	Data = 150h ⟶ (r6)(AG2)											
6	Data = 14Fh ⟶ (r7)(AG2)											
7	Data = 0 ⟶ (LS) of MAC1 and MAC2											
8	Data = 1 ⟶ (LSP)(SEQ)											
9	Data = label6 ⟶ RAM(1)(SEQ)											
10	Data = label7 ⟶ RAM(2)(SEQ)											
11	Data = $M_1 - 2$ ⟶ (c3)(SEQ)											
12	Data = N - 3 ⟶ (c2)(SEQ)											
label12 13	Data=N-3⟶ (c0)(SEQ) & (i0)(AG2)⟶(r0)(AG2) & (b0)(AG1) ⟶(r0)(AG1											
label1	Computation			MAC1			MAC2					
14	X	XD	Y		I/O	X	XD	Y		I/O	MEM	
14(1)	x_4					h_0						
15(2)		x_4	h_4				h_0	x'_0				
16(3)	x_5					h_1						
17(4)		x_5	h_3	$\Sigma = x_4 h_4$			h_1	0	$\Sigma' = h_0 x'_0$			
18(5)	x_6					h_2						
19(6)		x_6	h_2	$\Sigma = \Sigma + x_5 h_3$			h_2	0	Σ'			
20(7)	x_7					h_3						
21(8)		x_7	h_1	$\Sigma = \Sigma + x_6 h_2$			h_3	0	Σ'			
22(9)	Data = N - 2 is subtracted from (r2)(AG1)											
23(10)						1						
24(11)					Σ		1	Σ	$\Sigma' = \Sigma + \Sigma'$			
25(12)	Data = N is added to (r4)(AG2)											
26(13)			0					0	y_0			

27(14)	Continues to next sequential microprogram location											
28(15)											y_0	wr
29(16)	Rapeats the above computation for N - 1 times for all overlapping terms by jumping to label2 unconditionally											
78	Decrements (r2)(AG1)											
label4 79	(i0)(AG1) ⟶ (r0)(AG1) & (i0)(AG2) ⟶ (r0)(AG2) & Data = N-2 ⟶ (c1)(SEQ)											
label3 80(1)	h_0					h_0						
81(2)		h_0	x_7				h_0	x'_4				
82(3)	h_1					h_1						
83(4)		h_1	x_6	$\Sigma = h_0x_7$			h_1	x'_3	$\Sigma = h_0x'_4$			
84(5)	h_2					h_2						
85(6)		h_2	x_5	$\Sigma = \Sigma + h_1x_6$			h_2	x'_2	$\Sigma = \Sigma + h_1x_3$			
86(7)	h_3					h_3						
87(8)		h_3	x_4	$\Sigma = \Sigma + h_2x_5$			h_3	x'_1	$\Sigma = \Sigma + h_2x'_2$			
88(9)	h_4					h_4						
89(10)		h_4	x_3	$\Sigma = \Sigma + h_3x_4$			h_4	x'_0	$\Sigma = \Sigma + h_3x'_{11}$			
90(11)	Data = N - 1 is added to (r2)(AG1)											
91(12)			0	y_7				0	y_{12}			
92(13)	Data = N + 1 is added to (r4)(AG2)											
93(14)					y_7							wr
94(15)											y_{12}	wr
95	Repeats the above operation for M_1 times by jumping unconditionally to label4											

Results & Discussion

1 (i) [5×16] convolution ⟶ 215 cycles .

(ii) Throughput = $\frac{100}{10^{-7} \times 215}$ = 4.673 MOPs

2 (i) [10×30] convolution ⟶ 659 cycles

(ii) Throughput = $\frac{390}{10^{-7} \times 659}$ = 5.93 MOPs.

The speed-up factor in this example, comparing with System 1 is about 1.52-1.56 times.

4.4 META-ASSEMBLER

User should feed the microcoded system in binary format ,which is also called machine code . It is a laborious task , to change each field's instruction into the corresponding binary format . And hence a meta-assembler , which converts the microprogram sequence in assembly language into machine code , is

used here [2].

The meta-assembler works in two passes as shown in Fig 4.4 .

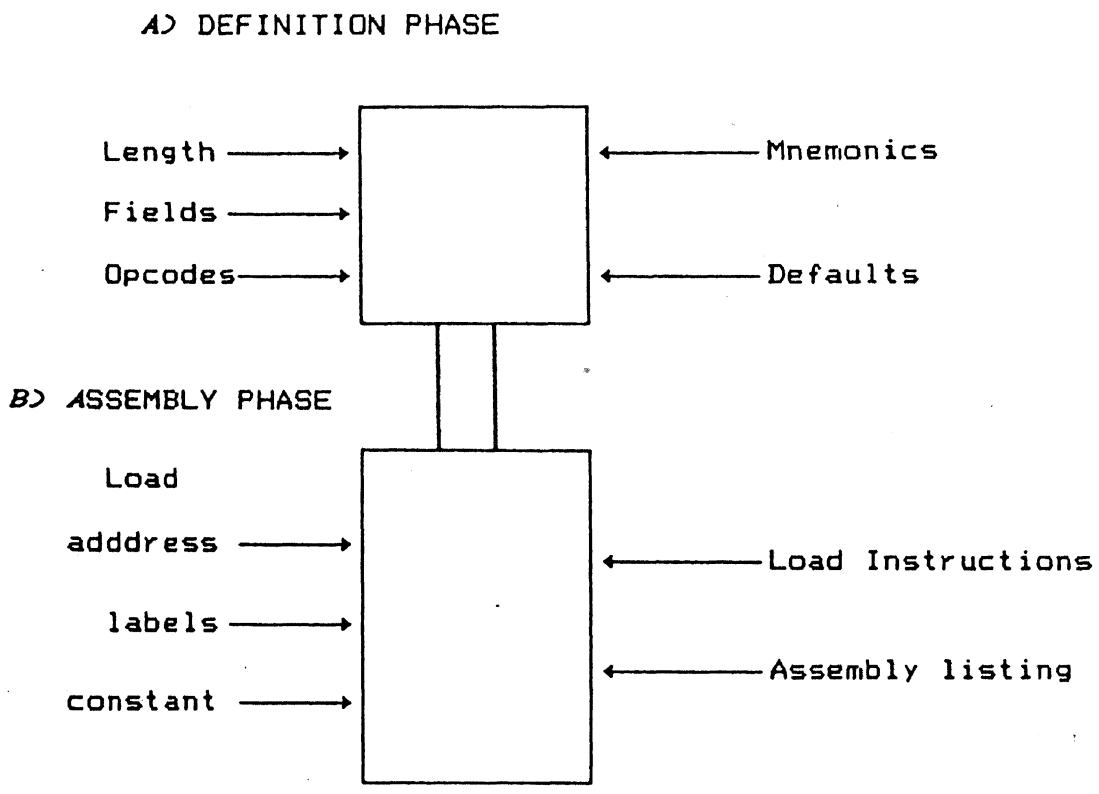


Fig 4.9 Two Phase Meta-Assembler

In the first phase , a definition file is processed to get a compact definition file . This compact definition file along with microprogram assembly language is procesed to get machine for the microprogram . The definition file and compact definition for each proposed system are given in Appendix D , E and F respectively .

CHAPTER 5

CONCLUSIONS

Microcoded systems are extensively used in signal processing applications to achieve high functional parallelism and thereby attaining high throughput.Simulation of a system is a software tool widely used for system verification and program development. In this thesis three microcoded signal procesors have been proposed and a simulator for these systems has been developed.

The simulator permits programs to be finely-tuned in software prior to making significant effort to bring up the target system in hardware. The simulator is interactive, it allows displays of registers and memory contents. Single stepping and full speed run is possible.

Using the simulator, the performance of these systems has been evaluated by executing matrix mltiplication and convolution algorithms.

It is observed that of all the systems, System 3 is the most efficient, achieving double the efficiency of System 1. This is because of its separate bus structure.

It is also found that the single-port structure of **1110A** processor considerably effects system throughput since all I/O operations take place via a single port. By replacing this processor by a multiport processor like **ADSP-1101** (which has two input-ports and one output port), and using a dual port memory for 2-processor systems, one can achieve a significant increase in the system throughput.

REFERENCES

1. DSP Product data book, Analog devices, 1987.

2. Wandhekar, " A Simulator for a Systolic Array Signal Processor (SASP) Based on ADSP-14XX and 32XX Chip Set" , M.Tech Thesis,1990.

3 Richard J. Higgins," Digital Signal Processors in VLSI", Prentice Hall , 1990.

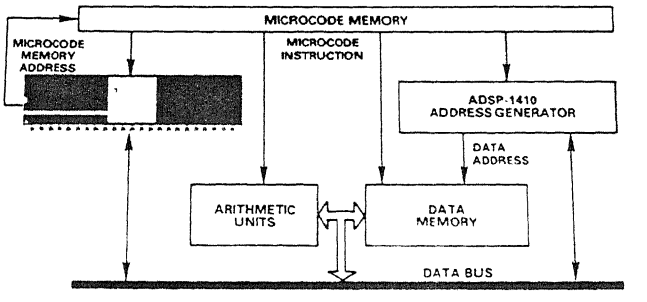
ANALOG DEVICES

Word-Slice™ Program Sequencer

FEATURES

16-Bit Microcode Addressing Capability
Look-Ahead™ Pipeline
Extensive Interrupt Processing, With Ten On-Chip Interrupt Vectors
70ns Cycle Time; 25ns Clock-to-Address Delay
64-Word RAM for Storing:
- **Subroutine Linkage**
- **Jump Addresses**
- **Counters**
- **Status Register**

375mW Maximum Power Dissipation with CMOS Technology
48-Pin DIP



WORD-SLICE™ MICROCODED SYSTEM WITH ADSP-1401

ENERAL DESCRIPTION

he ADSP-1401 is a high-speed microprogram controller op-nized for the demanding sequencing tasks found in digital gnal processors and general purpose computers. In addition to gh speed (25ns clock-to-address delay) and large addressing nge (64K of program memory), this Word-Slice component ıs unique features that make it highly versatile:

- on-chip storage and control of ten prioritized and maskable interrupts
- four decrementing event counters
- absolute, relative and indirect addressing capability
- download capability (writeable control store) and
- a dynamically configurable 64-word RAM.

ıe ADSP-1401 microprogram sequencer's main task is to ovide the appropriate microprogram addressing to support ogramming requirements (e.g., looping, jumping, branching, broutines, condition testing and interrupts). An internal Look-ıead pipeline, controlled by both phases of the clock, allows e ADSP-1401 to satisfy these requirements at very high speed.

uring each micro-instruction, the ADSP-1401 monitors the nditions and instructions to determine the next microprogram dress. This address can come from one of several sources: the ıck, the jump address space in the RAM, the data port, the terrupt vectors, or the microprogram counter. An extensive t of conditional instructions are also available, including jumps, anches, subroutines, interrupts, and writeable control store.

The ADSP-1401's internal 64-word RAM is user-configurable into three regions; subroutine stack, register stack and indirect jump address space. The subroutine stack is used for linking interrupts and subroutines and, during their execution, allow storage of system states. The register stack allows association of unique jump addresses with various levels of interrupts and subroutines (both local and global stacks are provided). Indirect jump capability is also supported, addressing for which is provided at the data port.

Interrupts are handled entirely on chip. The ADSP-1401's internal interrupt control logic includes registers for eight external (user) interrupt vectors, a mask register, and a priority decoder. Two additional vectors are reserved for internally-generated interrupts resulting from counter underflow and stack limit violation. A stack limit violation is caused by stack overflow, underflow or collision. A mechanism is provided for recovering from stack violations.

The ADSP-1401's four decrementing 16-bit counters are used to track loops and events. These counters generate a signal when negative. This negative condition is used by several conditional instructions and can also trigger an internal interrupt.

Two Technology Way; Norwood, MA 02062-9106 U.S.A.
Tel: 617/329-4700 Twx: 710/394-6577
Telex: 174059 Cables: ANALOG NORWOODMASS

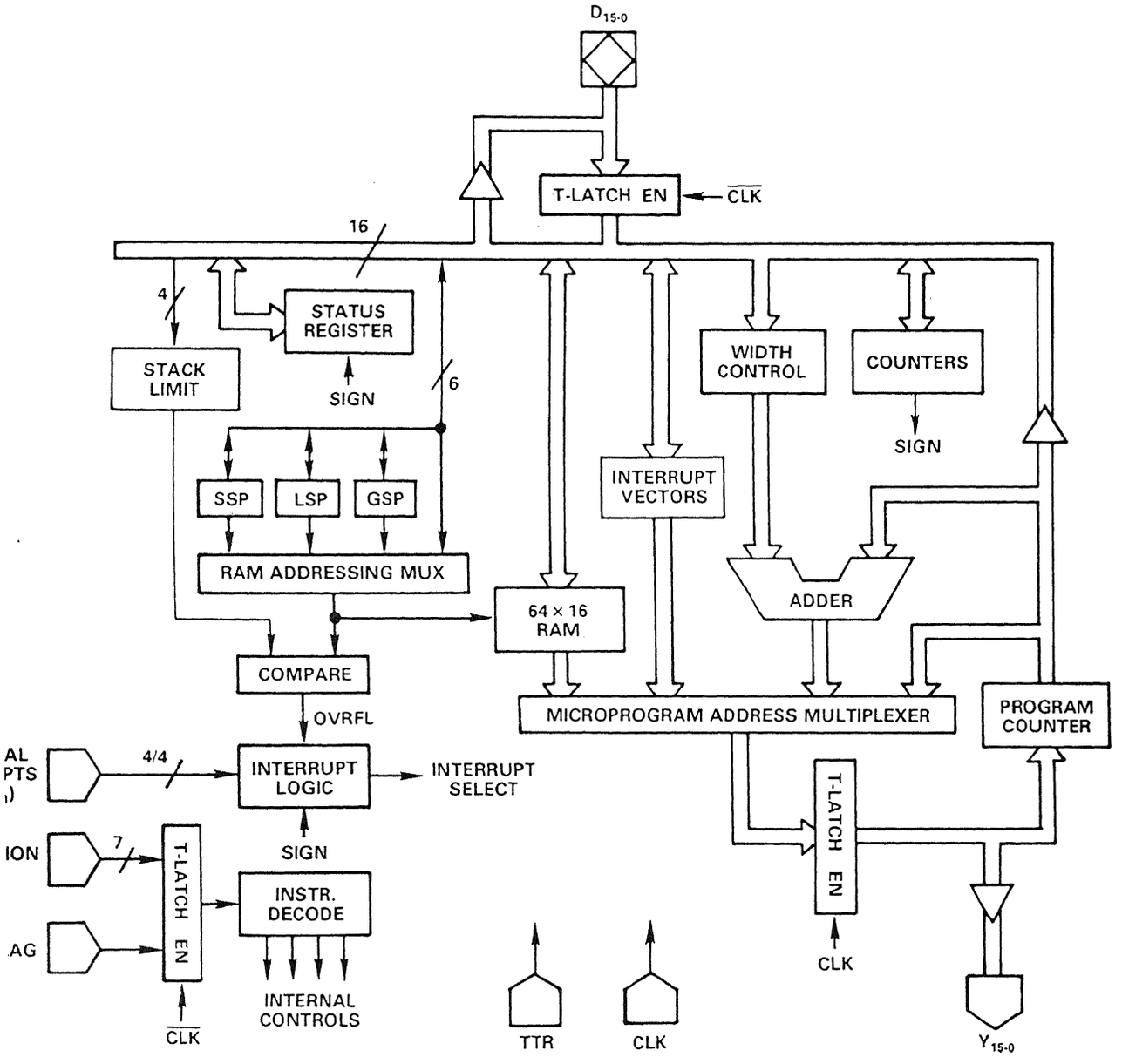


Figure 1. ADSP-1401 Block Diagram

ADDRESSING MODES

Direct: both absolute and relative
Indirect: from internal RAM

HARDWARE FEATURES

Instruction Port
Bidirectional Data Port
Four Input Address Multiplexer
Three Stack Pointers
Four Event Counters
Condition Flag
Eight Prioritized and Maskable User Interrupts
TTR Pin:
- Trap
- Three-State
- Reset

INSTRUCTION TYPES

Jumps and Branches
Stack Operations
Status Register Operations
Counter Operations
Interrupt Control
Relative Address Width Controls
Instruction Hold Control
Writeable Control Store
Dedicated Counter Underflow Interrupt
Dedicated Stack Overflow Interrupt

ADSP-1401 PIN ASSIGNMENTS

Pin Name	Description
I_6-I_0	The 7-bit microinstruction controlling the ADSP-1401.
$Y_{15}-Y_0$	Output bus which provides addresses to the microprogram memory.
$D_{15}-D_0$	Bidirectional Data bus for transferring data to or from the ADSP-1401.
$EXIR_{4-1}$	Four external interrupt request lines. Note that internal circuitry supports 8 interrupts with the aid of an external 2 to 1 multiplexer.
CLK	External clock input
FLAG	An input used for conditional instructions. Its source is usually a condition multiplexer.
TTR	A multi-purpose pin accommodating traps, output disable and reset.
V_{DD}	+5 Volt supply.
GND	Ground.

In a cache-based system, microcode is generally executed from the high-speed cache. If an access is attempted to code not resident in the cache area, the cache memory controller must detect the discrepancy and generate an exception to the access (a "cache miss"). Then, the missing code segment must be downloaded to the cache memory area (see: Instruction Set Description – Writeable Control Store, 2.7).

When a cache miss occurs, the cache memory control logic asserts the TTR pin while stretching the system clock LO. Upon detecting the trap request, the sequencer immediately generates the highest priority interrupt, IR_9, replacing the current address (that causing the cache miss). The cache miss address is pushed on the subroutine stack and popped after the interrupt service routine has reloaded the cache area with the missing code segment.

The trap interrupt differs from the standard interrupt protocol in three ways:

1. The interrupt vector, IV_9, is output asynchronously, i.e., it occurs t_{TRAD} after asserting the Trap signal and must occur *before* the next cycle! To accomplish this, a clock stretch cycle may be needed to allow enough time to fetch the new instruction.
2. The *current address* is pushed onto the SS for later restoration (after the cache miss is resolved), whereas standard interrupts push the *current address + 1*.
3. Trap interrupts *cannot* be masked or disabled. Note that if IR_9 is also used for stack overflow and underflow, the service routine must discriminate which actually occurred.

Caution: because trapping is asynchronous, spikes on the TTR pin wider than 3ns during clock LO may initiate inadvertent trapping.

Three-State
The address port is placed in a high-impedance state when the TTR pin is HI during clock HI and LO during clock LO. The TTR signal is latched during clock LO and transparent during clock HI. This facilitates full cycle, three-state control. (Note that the IDLE instruction can also place the address port in a high-impedance state.)

Reset
The TTR pin may be used to initialize the ADSP-1401 by asserting it (HI for both clock phases) for at least three full cycles. Use of the reset operation alone does not require the multiplexing described above. However, if the trap and/or three-state controls are also needed, they must not occur in the same cycle (this would be an abnormal situation), as this constitutes a reset. The RESET signal forces a zero output address, places the data port in the high-impedance state, and resets internal registers as follows:

Sequencer Status after RESET Operation

Parameter	Reset Condition
Program Counter	μCode Location 0000_{16}
Subroutine Stack Pointer (SSP)	RAM Location 00_{10}
Stack Limit Register (SLR)	RAM Location 32_{10}
RAM Data	No Change
Counters	No Change
Interrupt Mask (SR_{15-6})	All Bits to '0' (Unmasked)
Interrupt Vector File	No Change
Interrupt Vector Pointer (IVP)	Set to IRV_0
SR_{5-4}	'00' (16-Bit Relative Offsets)
SR_3	'0' (LSP Selected)
SR_2	'0' (Interrupts Disabled)
SR_1	'0' (Sign Bit Cleared)
SR_0	'0' (Latched Interrupt Mode)
Writeable Control Store Mode	Cleared

NOTE:
The first instruction (microcode location 0000_{16}) must be a "CONT".

2.0 INSTRUCTION SET DESCRIPTION

The instruction set is divided into seven categories pertaining to generic operation (see data sheet outline or Mnemonics and Opcodes, 4.5).

Several instructions employ two instruction bits (I_1 and I_0) to specify a counter (C_{3-0}) and/or a local register (R_{3-0}, relative to the RSP) as arguments. Nine of the conditional instructions use another two instruction bits (I_3 and I_2) to select one of the four condition modes:

'00' UNCONDITIONAL
'01' NOT FLAG
'10' FLAG
'11' SIGN

The sign bit of the status register, SR_1, may also be used to (implicitly or explicitly) store an external condition. This is useful if the condition results from an operation performed in the middle of a loop, but is not tested until the end; the loop is exited with an "If Sign: Jump" instruction. Recall that any subsequent counter operations will overwrite SR_1.

2.1 Jump and Branch Instructions

Jump and branch instructions provide flow control of microcode execution, offering three-way branches, jumps, subroutine calls, returns, and addressing mode selection (see Figure 6). These instructions support conditional control, allowing addressing from the register stack, the data port, or the indirect jump address space in the RAM. Generally, they are of the form:

If Condition: Do Operation; Else, Continue.

JPCOF IF FLAG: JUMP PC

The address is not incremented while the flag is at a logic HI, i.e., PC< = PC. If the flag is LO, the next address is (PC + 1).

JPCNF IF NOT FLAG: JUMP PC

The address is not incremented while the flag is at a logic LO, i.e., PC< = PC. If the flag is HI, the next address is (PC + 1).

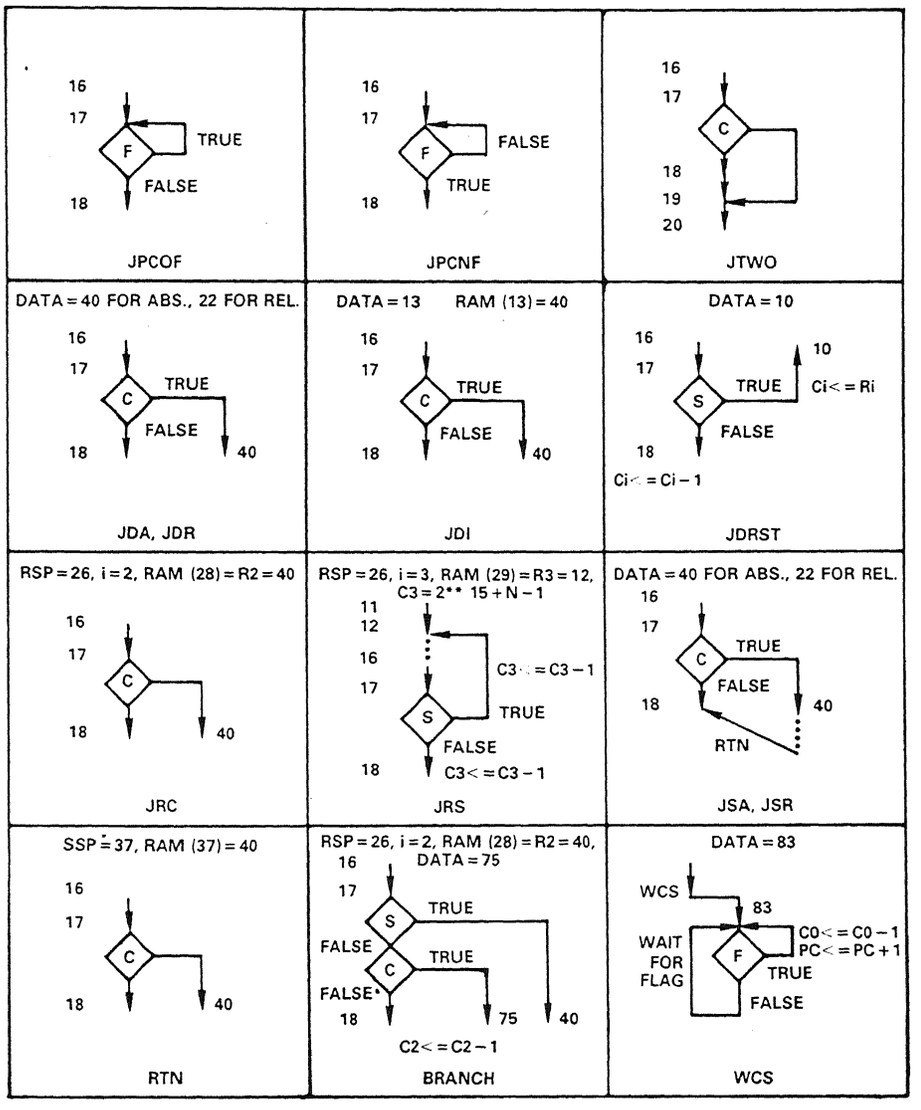


Figure 6. Instruction Flow Charts

TWO IF CONDITION: JUMP PC + 2

If the condition specified is met, this instruction causes the next sequential microprogram address to be skipped. This instruction allows single instruction bypassing or interleaving without need to provide explicit addressing.

JDA IF CONDITION: JUMP DATA, ABSOLUTE

If the specified condition is met, this instruction causes a jump to the absolute address at the data port. If the condition is not met, the next sequential instruction will be executed.

JDR IF CONDITION: JUMP DATA, RELATIVE

If the condition specified is met, the address at the data port will be added to the PC and output (jump distance is offset plus one). The offset width is determined by the address width selection (8, 12, or 16-bits). If the condition is not met, the next sequential instruction will be executed.

JDI IF CONDITION: JUMP DATA, INDIRECT

If the condition specified is met, this instruction will output the address stored in the RAM address given by bits D_{5-0} of the data port. If the condition is not met, the next sequential instruction will be executed.

JDRST IF SIGN OF C_i: JUMP DATA, $C_i <= R_i$;
ELSE, $C_i <= C_i - 1$

This instruction first tests the sign of the counter, C_i. If negative, the address at the data port is output and the counter is re-initialized (reset) with the data in the register pointed to by (RSP + i). If the sign is positive, the counter is decremented and the next sequential address is output. The register and counter use the same subscript, i.

JRC IF CONDITION: JUMP R_i. (COND ≠ SIGN)

If the condition specified is met, output the address in RAM at the location (RSP + i), where i is given by I_{1-0} of the instruction. The selected condition may not be SIGN, as this is the JRS instruction. The PC may be pushed on the register stack and referenced as a register thus allowing a "jump to stack" instruction which is useful for looping.

JRS IF SIGN OF C_i: JUMP R_i, $C_i <= C_i - 1$;
ELSE, $C_i <= C_i - 1$

This instruction first tests the sign of counter, C_i. If negative, output the address in RAM at location (RSP + i). If the sign is positive, the next sequential microprogram address is output. The counter is always decremented after the test.

JSA IF CONDITION: JUMP SUBROUTINE, ABSOLUTE

If the condition specified is met, the 16-bit absolute address at the data port is output and the PC will be pushed onto the subroutine stack. If the condition is not met, the next sequential instruction will be executed.

JSR IF CONDITION: JUMP SUBROUTINE, RELATIVE

If the condition specified is met, the address at the data port is added to the PC and output (jump distance is offset plus one) and the PC is pushed onto the subroutine stack. The offset width is determined by the address width selection (8, 12, or 16-bits). If the condition is not met, the next sequential instruction will be executed.

RTN IF CONDITION: RETURN FROM SUBROUTINE

This instruction is used to return from subroutines. If the condition specified is met, the subroutine stack is POPped, which outputs the return address and decrements the SSP. If the condition is not met, the next sequential instruction will be executed.

BRANCH IF SIGN OF C_i: JUMP R_i, $C_i <= C_i - 1$;
ELSE, IF CONDITION:
JUMP DATA, $C_i <= C_i - 1$;
ELSE, $C_i <= C_i - 1$ (COND ≠ SIGN)

This instruction implements a three-way branch with the address source from the data port, register R_i, or the PC. The instruction first tests the sign bit of the counter C_i; if negative, the output address is given by R_i, i.e., RSP + i. If the sign was not true, but the specified condition is true, the address source is the data port. If the sign was not true and the condition is not met, the next sequential instruction is executed.

The counter and the register use the same subscript value i. The counter is *always* decremented. Note that this instruction uses only absolute data addresses; relative addressing is not available with the three-way branch instruction.

2.2 Stack Operations

Subroutine Stack
Subroutine Stack Pointer (SSP) instructions are used for maintaining the subroutine stack. These instructions may also be used to upload or download the entire RAM for examination, stack expansion or context switches.

PSDSS PUSH DATA ONTO SS

Increments the stack pointer and then loads the RAM location specified by the SSP with the data at the data port.

PPSSD POP SS TO DATA PORT

Transfers the contents of the stack location given by the stack pointer to the data port and decrements the stack pointer.

WRSSP WRITE SSP

Loads the SSP with bits D_{5-0} of the data port.

RDSSP READ SSP

Read the 6-bit subroutine stack pointer. This allows the value of the stack pointer to be saved or examined. Bits D_{5-0} of the data port correspond to bits 5-0 of the SSP. The 10 MSB's of the data port (D_{15-6}) are undefined.

DSSP DECREMENT SSP

Decrements the stack pointer without reading.

Register Stack
Register Stack Pointer (RSP) instructions are used to upload and download the entire RAM for initialization, examination, or context switching and to maintain the RAM space allocated to local and global jump registers. As previously discussed, register stack instructions refer to either the Local Stack Pointer (LSP) or the Global Stack Pointer (GSP), depending upon the status register (SR_3). If SR_3 is LO, register stack instructions pertain to the LSP. If SR_3 is HI, register stack instructions pertain to the GSP.

SGSP SELECT GSP

Select the Global Register Stack Pointer. Set Status bit SR_3 (HI).

SLSP SELECT LSP

Select the Local Register Stack Pointer. Clear Status bit SR_3 (LO).

RDRSP READ RSP

Transfers the RSP to the data port bits D_{5-0} for examination or storage. The 10 MSBs (D_{15-6}) of the D port are undefined.

WRRSP WRITE RSP

Preload the selected RSP (LSP or GSP) with bits D_{5-0} of the data port.

PSPC PUSH PC ONTO RS

Decrements the RSP and writes the PC to the register stack. This instruction may be used to set up a JRC loop (IF CONDITION: JUMP R_i = PC).

PSGSP PUSH GSP ONTO SS

Increment the SSP and write the GSP onto the subroutine stack.

PPGSP POP GSP FROM SS

Write the subroutine stack to the GSP and decrement the SSP.

PSDRS PUSH DATA ONTO RS

Decrement the RSP and then write the data at the data port into the location specified by the updated RSP.

PPRSD POP RS TO DATA PORT

Transfers RAM data pointed to by the RSP to the data port and then increments the RSP.

AIRSP ADD i TO RSP

Add i to the register stack pointer. Note that i = 0, 1, 2, or 3 in this instruction corresponds to 4, 1, 2, or 3, respectively. This instruction effectively removes up to four registers from the stack.

S1RSP SUBTRACT ONE FROM RSP

Subtract 1 from the RSP without a write. This instruction is used to modify the RSP without explicitly reloading it.

S4RSP SUBTRACT FOUR FROM RSP

Subtract four from the RSP without a write. This instruction may be used to modify the RSP without explicitly reloading it.

2.3 Status Register Operations

The status register bits, SR_{15-0}, contain ten mask bits, SR_{15-6}, for masking interrupts IR_{9-0}, and six control bits, SR_{5-0} (see Bidirectional Data Port, 1.4). The entire status register can be read or written via the data port, or pushed or popped to/from the subroutine stack. Upon RESET, the entire status register is initialized to zero.

RDSR READ SR

The entire status register (SR_{15-0}) is output over the data port (D_{15-0}).

WRSR WRITE SR

Write the data port (D_{15-0}) to the status register (SR_{15-0}).

PSSR PUSH SR ONTO SS

Increment the SSP and then write the status register to the subroutine stack.

PPSR POP SR FROM SS

The top of the subroutine stack is written into the status register, and then the SSP is decremented.

2.4 Counter Operations

Counters may be pushed and popped to/from the subroutine stack or loaded directly from the data port. The counters may be read externally by pushing the counters onto the subroutine stack then popping the subroutine stack to the data port. The device has four counters, denoted C_i, which are indexed by the two LSB's of the instruction.

If a jump is required *after* N events (until sign), the counter should be loaded with two less than the number of events desired (N − 2). If a jump is required *for* N events (while sign), the counter is loaded with $2^{15} + N - 2 = 8000_{16} + N - 2$.

Care must be taken when using the counter underflow interrupt (IR_0, see 1.4.3) to clear the sign bit *before* the IR_0 mask bit is cleared.

WRCNTR WRITE C_i

Write to the selected counter, C_i, from the data port.

CLRS CLEAR SIGN BIT

Clear status register bit SR_1.

SETS SET SIGN BIT

Set status register bit SR_1.

PSCNTR PUSH C_i ONTO SS

Increment the SSP and write the specified counter onto the subroutine stack.

PPCNTR POP C_i FROM SS

Transfer the data from the subroutine stack to the counter specified by the instruction, then decrement the SSP.

DCCNTR DECREMENT C_i

Unconditionally decrement counter C_i.

IFCDEC IF CONDITION: DECREMENT C_0

Decrement counter C_0 on condition. The status register bit SR_1 is used if the sign condition is selected.

2.5 Interrupt Control

Detailed interrupt operation is described in the Interrupts section (1.4.3). Here, specific interrupt operations such as interrupt clearing, IRV read/write, interrupt mask manipulation, etc., are described.

CCIR CLEAR CURRENT INTERRUPT

Allows nesting of user interrupts IR_{8-1} on subsequent instructions by clearing both the interrupt latch bit currently being serviced and the interrupt in progress signal (IRIP), re-enabling interrupts. If an external interrupt is pending, the associated IR vector will not be output until the cycle following CCIR. Internal interrupts (IR_9 and IR_0) are *not* cleared by CCIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

CAIR CLEAR ALL INTERRUPTS

Clears external interrupt latches IR_{8-1}, and re-enables the interrupt interface (IRIP cleared LO). The next sequential instruction will be executed prior to the jump to a pending interrupt. Internal interrupts (IR_9 and IR_0) are *not* cleared by CAIR and must be explicitly cleared through the SLRIVP and CLRS instructions, respectively.

RTNIR RETURN FROM INTERRUPT

Clears the current interrupt latch for IR_{8-1}, re-enables interrupts (IRIP cleared LO), and pops the return address from the subroutine stack. The next sequential instruction will be executed prior to the jump to a pending interrupt routine. Internal interrupts are *not* cleared and the IR_9 and IR_0 interrupt latches must be cleared explicitly through the SLRIVP and CLRS instructions, respectively.

RDIV READ IRV AND INCREMENT IVP

Outputs the interrupt vector currently pointed to by IVP to the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

WRIV WRITE IRV AND INCREMENT IVP

Writes the interrupt vector currently pointed to by the IVP from the data port and then increments the IVP. Interrupts should be disabled when writing or reading interrupt vectors.

IRMBC IR MASK BITWISE CLEAR

Allows selected IR mask bits to be cleared. Data port bits D_{15-6} are applied to status register bits SR15-6 (corresponding to mask bits for IR_{9-0}). Those data bits which are HI will clear the mask bit, while those data bits which are LO will leave the mask bit intact. Data port bits D_{5-0} are ignored.

IRMBS IR MASK BITWISE SET

Allows selected IR mask bits to be set. Data port bits D_{15-6} are applied to status register bits SR_{15-6} (corresponding to mask bits for IR_{9-0}). Those data bits which are HI will set the mask bit, while those data bits which are LO will leave the mask bit intact. Data port bits D_{5-0} are ignored.

DISIR DISABLE INTERRUPTS

Disables the execution of all further interrupts by clearing the enable interrupt flag (SR_2). External interrupts continue to be latched.

ENAIR ENABLE INTERRUPTS

Enables execution of interrupts by setting the enable interrupt flag (SR_2).

SLIR SELECT LATCHED INTERRUPTS

Places the interrupt request latches in the latched mode for interrupts IR_{8-1} (SR_0 LO). Interrupts are latched if they are valid at the appropriate clock edge. Interrupts IR_{8-5} are latched at the positive going clock edge while IR_{4-1} are latched at the negative going clock edge.

STIR SELECT TRANSPARENT INTERRUPTS

Places the interrupt request latches in the transparent mode (SR_0 HI) for interrupts IR_{8-1}. The interrupt request is only valid while the external interrupt inputs are high. Interrupts are still processed on the next cycle, so long as they meet the minimum interrupt setup specification. Note that selecting transparent interrupting will clear any pending interrupts stored in the interrupt latch.

SLRIVP WRITE SLR WITH D_{5-2}, AND IVP WITH D_{15-12}

Loads the 4-bit stack limit register (SLR) and the 4-bit interrupt vector pointer (IVP) from the data port. This instruction also clears the stack overflow interrupt request IR_9.

For stack overflow detection, the active 6-bit stack pointer (SSP, LSP or GSP) is compared to a 6-bit word comprised of the 4-bit SLR (MSBs) and the two LSBs determined by the instruction type, as follows:

'00' for subroutine stack push (PSDSS); or,
'11' for register stack push (PSDRS).

For example, if a stack limit of 36_{10} and positioning of the IVP at IRV_7 is desired, the value '0111xxxxxx1001xx' is provided at the data port. Note that the SLR and IVP cannot be read.

The interrupt vector pointer (IVP) addresses the vector file for reading or writing interrupt vectors. To write interrupt vectors IRV_{9-0}, the IVP must first be initialized by SLRIVP. The WRIV instruction (see above) is then used to write the interrupt vector pointed to by the IVP, which is then incremented automatically.

2.6 Relative Address Width Controls

The width control instructions allow reduction of microcode when Jump Data Relative and Jump Subroutine Relative instructions need less than the full, 16-bit range. Use these instructions to sign extend the 8, 12 or 16-bit wide jump data presented at the data port. The jump width may be selected the explicit instructions or by directly setting the status regi bits SR_{5-4} as described below. Any of these three instructio will reset the Instruction Hold Control mode (see Misc. Inst tions – IHC, 2.7).

Note that selection of 8-bit width can be made with or with IHC. For all relative jumps, the jump distance is the offset

REL16 SELECT 16-BIT RELATIVE JUMPS

Select the 16-bit relative jump. This adds D_{15-0} at the data to the PC to obtain the jump address. The status bits SR_{5-4} set to '00'.

REL12 SELECT 12-BIT RELATIVE JUMPS

Selects the jump data from D_{11-0}. The offset is sign-extend allowing relative jumps in the range +2047 to −2048. The status bits SR_{5-4} are set to '11'.

REL8 SELECT 8-BIT RELATIVE JUMPS

Selects the jump data from D_{7-0}. The offset is sign-extende allowing relative jumps in the range +127 to −128. The st bits SR_{5-4} are set to '01'.

2.7 Miscellaneous Instructions

CONT CONTINUE

Increment and output the next location in microcode memc without any other changes. Allows straight line microcode execution.

IDLE DISABLE OUTPUTS AND JUMP PC

Places the address port into the high-impedance state, inhil program counter (PC) increments. Useful in applications w multiple sequencers share a common microcode address bu

This instruction causes the ADSP-1401 to behave as if the had stopped. The IDLE instruction may be latched interna by using IHC, freeing microcode for use by another device Note that while idle, external interrupts will continue to be registered and should therefore be masked or disabled.

IHC ENABLE INSTRUCTION HOLD CONTI

Sets SR_{5-4} to '10' and redefines the function of IR_1 to allo subsequent instruction to be held for repeated execution, rega of the instruction port. Use of the IHC mode requires that mask bit for IR_1 be set. See Instruction Hold Control, app 4.1 for more details.

While in the IHC mode, asserting IR_1 HI (prior to the sec half-cycle of any instruction) will hold that instruction anc disable *all* interrupts (although they continue to be latche until IR_1 is brought LO again (again, prior to the second ha

It is recommended that IR_1 be dedicated to control of the IHC mode (if needed). However, if it must also be used for subsequent interrupting, then the CAIR instruction should be executed before unmasking IR_1 (to clear the interrupt request resulting from use of IR_1 as the IHC control).

Use of IHC is constrained to 8-bit relative addressing (see Relative Address Width Controls, 2.6) and clearing IHC is accomplished by executing any of the relative address width control instructions (changing status register bits SR_{5-4}).

WCS WRITE CONTROL STORE

Provides sequential addressing during microcode downloads to a RAM based microcode store. The instruction may be interpreted as:

JUMP DATA;
IF FLAG: DECREMENT C_0 AND CONTINUE UNTIL INTERRUPTED.

Upon initiation of the WCS instruction, the sequencer outputs the address found at the data port (that of the first instruction to be downloaded). The external flag is then used to gate subsequent sequential addressing for the download and decrementing of counter C_0. This action continues until an interrupt is detected (from either a C_0 underflow, externally or the chip is RESET). Instructions at the instruction port are *ignored* during WCS, until the interrupt or reset occurs.

The external flag allows synchronization of an external memory with the sequencer. FLAG should be asserted HI as each new μcode word is made available for writing to μcode memory.

Notes on Using a Writeable Control Store:

- If a counter interrupt is desired, counter C_0 must be initialized with *two* less than the length of microcode segment to be downloaded.
- If counter interrupting is to be used to exit the WCS mode, IRV_0 should be unmasked and initialized with the address of the instruction to be executed upon WCS completion (see Interrupts, 1.4.3 for timing).
- Since interrupting is used to exit the WCS mode, the last address downloaded is pushed onto the SS stack as an interrupt return address. However, because it is not actually a return address, the SS should be popped immediately by decrementing the SSP (DSSP) to clear it of this last address.
- Since FLAG is used to gate the download, it should not become active until after the WCS instruction is executed.

See application note "Writeable Control Store using the ADSP-1401".

3.0 SPECIFICATIONS

This section describes the ADSP-1401's performance parameters. The Specifications Table lists the device's relevant electrical and switching characteristics, while Figure 7 presents the corresponding timing diagram.

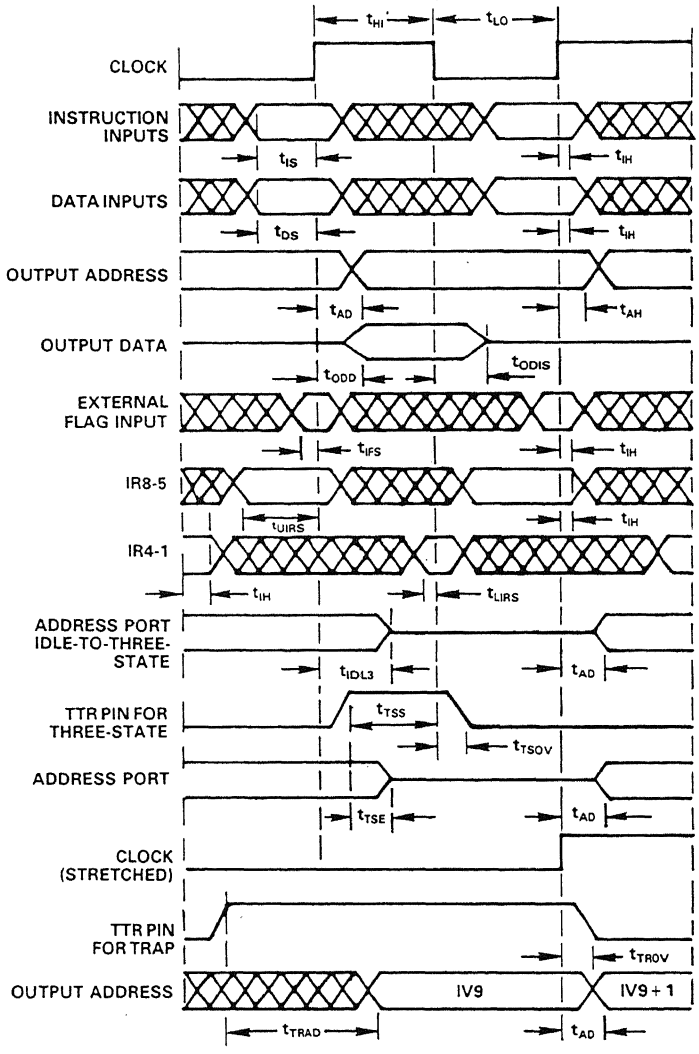


Figure 7. ADSP-1401 Timing Diagram

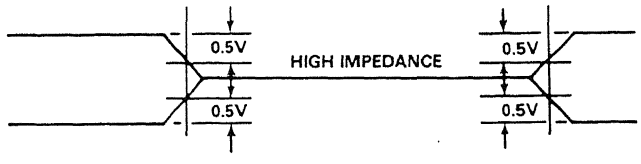


Figure 8. Three-State Reference Levels

ORDERING INFORMATION

Part Number	Temperature Range	Package
ADSP-1401JN	0 to +70°C	48-Pin Plastic DIP
ADSP-1401KN	0 to +70°C	48-Pin Plastic DIP
ADSP-1401JD	0 to +70°C	48-Pin Ceramic DIP
ADSP-1401KD	0 to +70°C	48-Pin Ceramic DIP
ADSP-1401SD	−55°C to +125°C	48-Pin Ceramic DIP
ADSP-1401TD	−55°C to +125°C	48-Pin Ceramic DIP
ADSP-1401SD/+	−55°C to +125°C	48-Pin Ceramic DIP
ADSP-1401TD/+	−55°C to +125°C	48-Pin Ceramic DIP
ADSP-1401SD/883B	−55°C to +125°C	48-Pin Ceramic DIP
ADSP-1401TD/883B	−55°C to +125°C	48-Pin Ceramic DIP

In the case of the program sequencer, for an external load capacitance of 50pF and a measured slew rate of 0.6V/ns, the peak current will be about 30mA. Since there are 16 such drivers, the total peak current may approach 480mA!

The internal ground and supply lines may undergo a large disturbance during this transition unless the ADSP-1401 is tied to a solid ground plane and good high frequency decoupling is used (0.1μF ceramic between GND and V_{DD} as close as possibl to the device). Otherwise, is it possible that internal data in the ADSP-1401 may be lost.

4.5 Mnemonics and Opcodes

Opcode bits "ii" select the relevant register (R_{3-0}) and/or counter (C_{3-0}). Opcode bits "cc" select the condition to be applied:

'00' UNCONDITIONAL
'01' NOT FLAG
'10' FLAG
'11' SIGN

The SIGN condition is precluded from instructions prefixed with "*".

Mnemonic	Opcode (I_{6-0})	Description
Jump and Branch Instructions:		
JPCOF	001 0101	IF FLAG: JUMP PC (*self*)
JPCNF	011 0101	IF NOT FLAG: JUMP PC (*self*)
JTWO	101 cc01	IF COND: JUMP PC+2 (*skip*)
JDA	111 cc11	IF COND: JUMP DATA, ABSOLUTE
JDR	111 cc01	IF COND: JUMP DATA, RELATIVE
JDI	101 cc10	IF COND: JUMP DATA, INDIRECT
JDRST	100 11ii	IF SIGN OF C_i: JUMP DATA, $C_i <= R_i$; ELSE, $C_i <= C_i - 1$
*JRC	110 ccii	IF COND: JUMP R_i
JRS	110 11ii	IF SIGN OF C_i: JUMP R_i, $C_i <= C_i - 1$
JSA	111 cc00	IF COND: JUMP SUB, ABSOLUTE
JSR	111 cc10	IF COND: JUMP SUB, RELATIVE
RTN	101 cc11	IF COND: RETURN FROM SUB
*BRANCH	100 ccii	IF SIGN OF C_i: JUMP R_i; ELSE, $C_i <= C_i - 1$, IF COND: JUMP DATA
Stack Operations:		
Subroutine Stack		
PSDSS	001 1110	PUSH DATA ONTO SS
PPSSD	011 1110	POP SS TO DATA PORT
WRSSP	000 1110	WRITE SSP
RDSSP	010 1100	READ SSP
DSSP	000 0010	DECREMENT SSP
Register Stack		
SGSP	000 0111	SELECT GSP
SLSP	000 0110	SELECT LSP
RDRSP	010 1111	READ RSP
WRRSP	000 1100	WRITE RSP
PSPC	010 0011	PUSH PC ONTO RS
PSGSP	000 0101	PUSH GSP ONTO SS
PPGSP	000 0100	POP GSP FROM SS
PSDRS	001 1111	PUSH DATA ONTO RS
PPRSD	011 1111	POP RS TO DATA PORT
AIRSP	010 10ii	ADD i TO RSP
S1RSP	000 1111	SUBTRACT 1 FROM RSP
S4RSP	011 1100	SUBTRACT 4 FROM RSP

Status Register Bit Assignments

Bit#	Function (HI/LO)
SR_{15}	IR_9 Mask Bit
•	•
•	•
•	•
SR_6	IR_0 Mask Bit
SR_{5-4}	Relative Jump Width Selection: '00' = 16-bit relative address width '01' = 8-bit width '10' = IHC Mode (8-bit width) '11' = 12-bit width
SR_3	Select GSP/LSP
SR_2	Enable/Disable Interrupts
SR_1	Set/Clear Sign Bit
SR_0	Select Transparent/Latched Interrupts

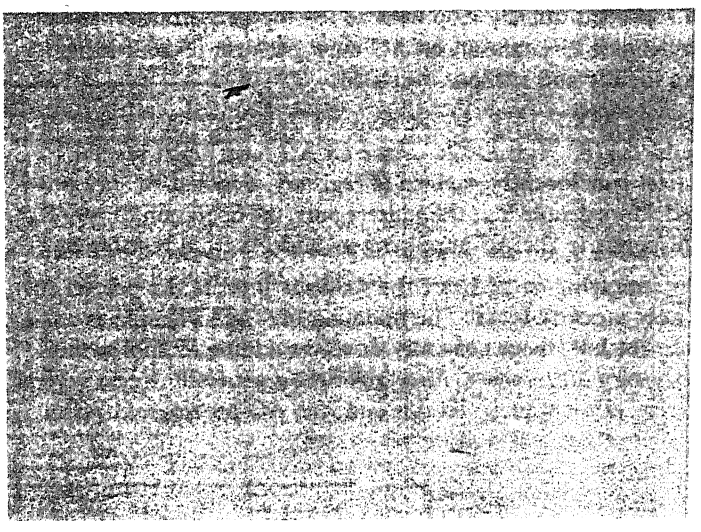
Status Register Operations:		
RDSR	010 1110	READ SR
WRSR	001 1100	WRITE SR
PSSR	010 0001	PUSH SR ONTO SS
PPSR	010 0010	POP SR FROM SS
Counter Operations:		
WRCNTR	011 10ii	WRITE C_i
CLRS	001 0100	CLEAR SIGN BIT
SETS	011 0100	SET SIGN BIT
PSCNTR	000 10ii	PUSH C_i ONTO SS
PPCNTR	001 10ii	POP C_i FROM SS
DCCNTR	011 00ii	DECREMENT C_i
IFCDEC	101 cc00	IF COND: DECREMENT C_0
Interrupt Control:		
CCIR	001 0001	CLEAR CURRENT INTERRUPT
CAIR	000 0001	CLEAR ALL INTERRUPTS
RTNIR	000 0011	RETURN FROM INTERRUPT
RDIV	010 1101	READ INTERRUPT VECTOR AND INCREMENT IVP
WRIV	000 1101	WRITE INTERRUPT VECTOR AND INCREMENT IVP
IRMBC	001 0011	IR MASK BITWISE CLEAR
IRMBS	001 0010	IR MASK BITWISE SET
DISIR	001 0110	DISABLE INTERRUPTS
ENAIR	011 0110	ENABLE INTERRUPTS
SLIR	001 0111	SELECT LATCHED INTERRUPTS
STIR	011 0111	SELECT TRANSPARENT INTERRUPTS
SLRIVP	001 1101	WRITE SLR<=D_{5-2} AND IVP<=D_{15-12}
Relative Address Width Controls:		
REL16	010 0100	SELECT 16-BIT RELATIVE ADDRESSING
REL12	010 0111	SELECT 12-BIT RELATIVE ADDRESSING
REL8	010 0110	SELECT 8-BIT RELATIVE ADDRESSING
Miscellaneous Instructions:		
CONT	000 0000	CONTINUE
IDLE	001 0000	IDLE
IHC	010 0101	ENABLE INSTRUCTION HOLD CONTROL
WCS	010 0000	WRITE CONTROL STORE

ANALOG DEVICES

Address Generato

FEATURES

- **16-Bit Addresses with Higher Precision Options**
- **High Speed, Clock-to-Valid-Address Delay of 20ns**
- **Look-Ahead™ Pipeline**
- **Versatile Addressing Hardware:**
  - **30 16-Bit Registers**
  - **16-Bit ALU with Left/Right Shift & Carry I/O**
  - **Comparator**
  - **Bit Reverser**
- **Dual Ports**
- **Powerful Single-Cycle Looping Instructions**
- **175mW Maximum Power Dissipation with CMOS Technology**
- **48-Pin DIP**



GENERAL INFORMATION

The ADSP-1410 is a fast, flexible address generator optimized for digital signal/array processors and other high-performance computers. This low-power CMOS device rapidly generates the data memory addresses required by routines such as digital filters, FFTs, matrix operations, and DMAs. With its 16-bit architecture, registers, dual ports, and speed, the 48-pin ADSP-1410 improves performance and reduces board space substantially, relative to bit-slice solutions.

The ADSP-1410's architecture features a 16-bit ALU, a comparator, and 30 16-bit registers. The registers are organized into four files: sixteen address (R) registers, six offset (B) registers, four compare (C) registers, and four initialization (I) registers.

The ADSP-1410 rapidly executes key address generating operations. In a single instruction cycle, the device can:

- output a 16-bit memory address;
- modify this memory address; and,
- detect when the address value has moved to or beyond a pre-set boundary and conditionally loop back to the top of a circular buffer.

Consequently, circular buffers and modulo addressing for data memories can be implemented without overhead.

The ADSP-1410's 10-bit microcode instructions include commands for looping, register read/writes, internal data transfers, and logical/shift operations. Instructions are normally supplied from an external source. However, an internal Alternate Instruction Register (AIR) can provide the instruction under external control, allowing microcode to be conserved in many applications.

The ADSP-1410 has a 16-bit address (Y) port for outputting addresses and a 16-bit data (D) port for I/O between internal and external registers. Also, an internal path allows external data, provided via the D port, to serve as an ALU source and to be directly output over the Y port for a DMA capability.

Double-precision (30-bit), single-cycle addressing can be performed by cascading two ADSP-1410's, with the MSB of each chip's D and Y port dedicated to interchip communication. Alternatively, a single AG can provide double-precision addres at a rate of one per two clock cycles.

The Look-Ahead™ pipeline eliminates the need for an exter microcode pipeline register by internaly latching instructions and addresses; microcode bits may be directly routed to the ADSP-1410 from microcode memory. Logically, the Look-Ahead™ pipeline is split into two havles: the first, located at the instruction (and data) port; and the second, located at the address port. Each half of the pipeline (input vs. output) has transparent latch which operates out of phase with the other: the address latch is transparent during the first half of the cy (clock HI); while the input latches (instruction and data) are transparent during the second half of the cycle (clock LO). T complementary arrangement allows new instructions to be decc (in preparation for the following cycle) while the program add for the current cycle is held steady.

Route 1 Industrial Park; P.O. Box 280; Norwood, Mass.
Tel: 617/329-4700 **TWX: 710/394**
West Coast **Mid-West** **Tex**
714/641-9391 **312/653-5000** **214/23**

ADSP-1410 OVERVIEW

Digital Signal Processing (DSP) and array processing systems require fast, flexible address generation circuitry. An Address Generator (AG) supplies the address of a location in data or coefficient memory. The value residing at the specified address is fetched and fed to an arithmetic unit for processing. The AG must then modify the address pointer in anticipation of the next data fetch. For algorithms that repetitively loop through data buffers, the AG may need to compare the address to a buffer end and conditionally loop back to the top of the buffer. Finally, to maximize throughput, an AG must perform its addressing tasks rapidly and without overhead.

With the ADSP-1410, 16-bit pointers to memory are stored in an address (R) register file. Since an AG must track several pointers concurrently, sixteen R registers, denoted R_n, are provided. If we denote Y as the address port, the operation "$Y \leftarrow R_n$" corresponds to the AG supplying an address from register R_n.

After supplying an address, the AG must update the pointer for the next memory fetch. The updating may be as simple as an increment but, more generally, involves adding or subtracting an arbitrary offset value. Also, algorithms generally access several different offset values. To this end, the AG provides six offset registers, denoted B_m, and can execute in a single-cycle the core operation:

$$Y \leftarrow R_n;\ R_n \leftarrow R_n + B_m.$$

In DSP applications, data arrays are often addressed as circular buffers. That is, when addressing reaches the buffer end, it wraps back to the beginning of the buffer. To implement this looping, the AG compares the supplied address to one of four compare registers, denoted C_j. If the address has moved to or beyond the end of the boundary ($R_n \geq C_j$), the device can transfer an initialization register value, denoted I_j, to the register ($R_n \leftarrow I_j$); otherwise, it is updated in normal fashion ($R_n \leftarrow R_n + B_m$). To minimize overhead, the AG can execute normal updates while also performing conditional re-initializations; again, in one core operation:

$$Y \leftarrow R_n;\ \text{IF } (R_n \geq C_j)\text{: } R_n \leftarrow I_j;\ \text{ELSE } R_n \leftarrow R_n + B_m.$$

Since the above instruction handles the looping required of circular buffer addressing, it is termed a looping instruction. To a large extent, the ADSP-1410's architecture and instruction set revolve around efficient implementation of this instruction. However, many variations of this instruction are supported on the device and spelled out in the following sections.

ADDRESS SOURCES
- Sixteen internal R registers
- External data provided over the D port

OFFSET SOURCES
- Six internal B registers
- Data Port

OFFSET OPERATIONS
- Increment ($R_n \leftarrow R_n + 1$)
- Decrement ($R_n \leftarrow R_n - 1$)
- Add Offset ($R_n \leftarrow R_n + B_m$)
- Subtract Offset ($R_n \leftarrow R_n - B_m$)
- Single-Bit Left/Right Shifts
- Logical Operations (AND, OR, XOR)

CONDITIONAL RE-INITIALIZATION
- Independent Inhibit/Enable for each of four initialization registers
- Conditional AIR execution (used for true modulo addressing)

OUTPUT/UPDATE SEQUENCE
- Normal (Pre-Update) Mode (output the address before update)
- Post-Update Mode (output the address after update)

PRECISION
- Single chip supplies 16-bit addresses
- Two chips cascaded provide 30-bit addresses
- One chip provides 30-bit addresses in two cycles

ADSP-1410 PIN ASSIGNMENTS

PIN NAME	DESCRIPTION
$Y_{15} - Y_0$	The address (Y) output port. In single-chip/double-precision mode, the MSB (Y_{15}) indicates whether the supplied address is the MSW or LSW (see Precision Modes). In two-chip/double-precision mode, the MSB conveys the carry/shift bit from the Least Significient (LS) to the Most Significient (MS) chip.
$D_{15} - D_0$	The bi-directional data (D) port. In two-chip/double-precision addressing mode, the MSB (D_{15}) of this port conveys CMP status from the partner chip.
$I_9 - I_0$	The instruction port.
CMP/Z	A dual function pin. Looping instructions, which compare address register values to compare register values, assert this pin HI to convey CMP status if i) $R \geq C$ for positive offsets, or ii) $R \leq C$ for negative offsets. Logical/Shift instructions assert this pin HI to convey the ZERO status of the result.
DSEL	Data Select control. Asserting this control HI causes data set up on the data port to substitute for the R value specified in the instruction.
AIR Enable	Alternate Instruction Register control. Asserting this control HI causes the device to execute an instruction stored in the internal AIR, rather than the instruction set up on the instruction port.
CLK	Clock
V_{dd}	+5 Volt Power Supply
GND	Ground

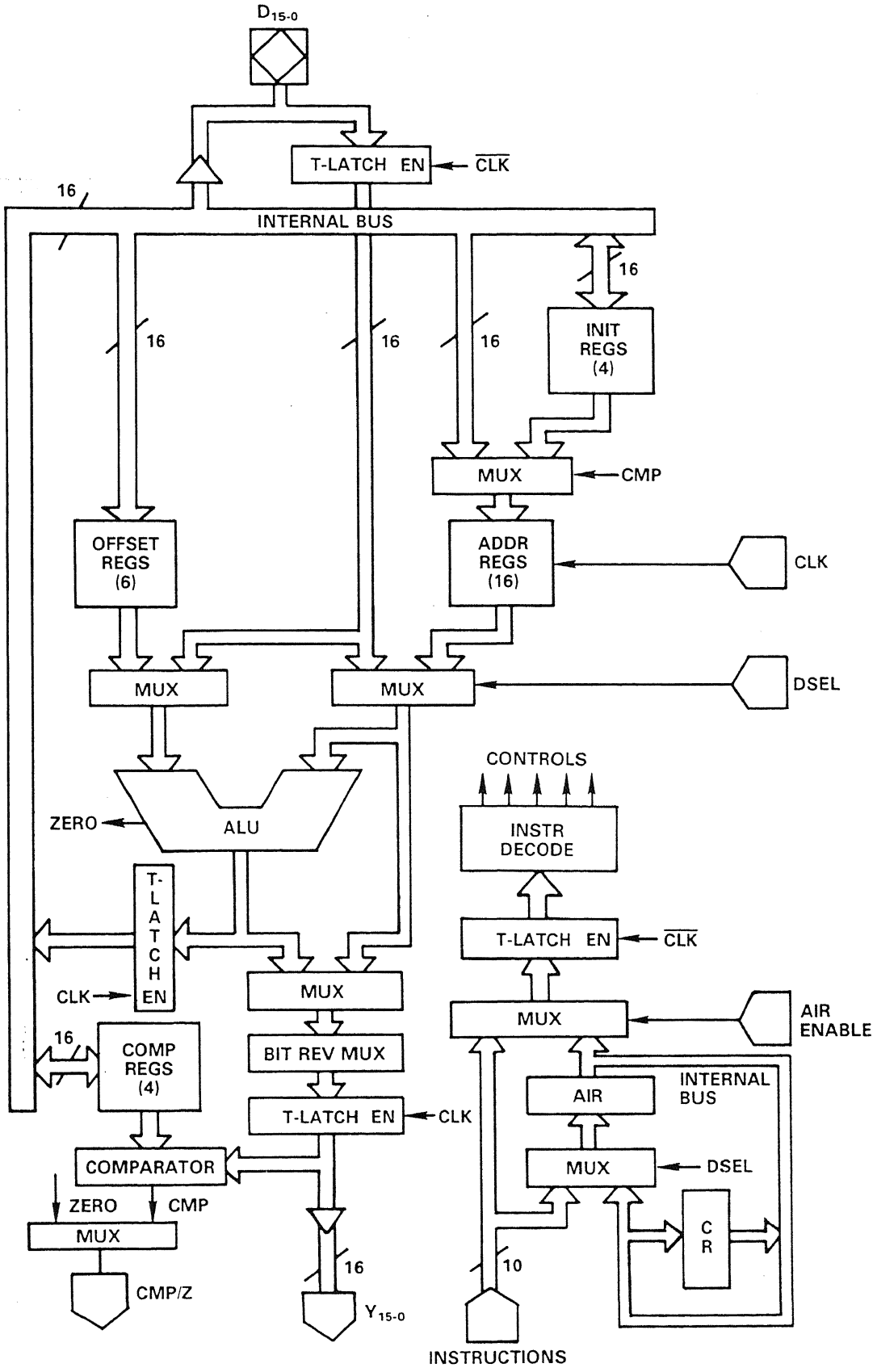


Figure 1. ADSP-1410 Functional Block Diagram

Latched Mode
(CR_6 LO). In latched mode, output values are enabled during phase one and latched at the address (Y) port during phase two.

Use of the latched mode guarantees that outputs remain stable throughout the current cycle (barring T_{AD}) regardless of changes at the instruction port. This, in contrast to the transparent mode, in which such changes may occur quickly enough to alter the output before cycle end.

Post-Update Mode
(CR_9 HI). Addresses are output after the update operation. The delay between the start of phase one and output of a valid address is extended in this mode to allow for updating. The addresses output are equivalent to the values written back into the specified address (R) register. In this mode, external data may be brought on chip, modified and output—in a single clock cycle.

Pre-Update Mode
(CR_9 LO). This is the normal update mode in which addresses are output over the address (Y) port prior to update operations (increment, decrement, offset, shift, and logical)—allowing addresses to be generated at maximal speed. Note however, that this mode requires two cycles to bring external data on chip, modify it, and supply it as an address.

Conditional AIR Execute Mode
(CR_{10} HI). In this mode, a valid CMP flag on looping instructions causes the next instruction to be executed from the AIR. The MODULO ADDRESSING section highlights a particularly valuable use of this mode.

Note that conditional re-initialization of address registers is disabled when using the conditional AIR execute mode, although routine updates (INC, DEC, ADD, and SUB) are still performed in accord with the instruction under execution (be it from the instruction port or the AIR).

(CR_{10} LO). Conditional AIR execution is disabled. Conditional re-initialization is fully operational, contingent upon the re-initialization mask (CR_{3-0}).

Table III summarizes the different ways the CMP status affects operation of the AG as a function of the conditional AIR execute mode control bit, CR_{10}, and the re-initialization mask, CR_{3-0}.

CMP STATUS	CR_{10} LO		CR_{10} HI
	CR_j LO	CR_j HI	
LO	No Effect	No Effect	No Effect
HI	CMP/Z goes HI	CMP/Z goes HI; $R_n \leftarrow I_j$	CMP/Z goes HI; Next instr. executed from AIR

Table III. Effect of Compare (CMP) Status for Looping Instructions; Note: j=3–0, the Re-Initialization Mask.

INSTRUCTION SET DESCRIPTION

The ADSP-1410's instruction set is partitioned into six groups, which are discussed below. First, however, issues spanning several instruction groups are discussed.

Most of the instruction groups contain instructions using one c the chip's six offset (B) registers. Without exception, these instructions have just two bits available for selecting the B regist Consequently, offset registers are partitioned into two banks. The upper/lower bank selection is maintained in the control register (CR_8) and is set or cleared by dedicated instructions. Whenever the "fourth" B register of either bank is specified (B_3 or B_7), the ALU's offset source becomes external data (see Table IV).

CR_8 & TWO-BIT OFFSET (B) REGISTER FIELD	OFFSET SOURCE
0 00	B0
0 01	B1
0 10	B2
x 11	Data Port*
1 00	B4
1 01	B5
1 10	B6
x 11	Data Port*

Table IV. Offset Value Structure

*Explicit use of DSEL is unnecessary when using B_3 or B_7 offsets; the of data is sourced from the data bus by default.

In several instruction groups, address (R) registers are used. all cases, asserting the DSEL pin allows external data to be substituted for an R value as both output and update data.

Two instruction groups (looping and logical/shift) both supply and update the address. Normally, addresses are supplied prior to updating (pre-update). In post-update mode however, the addresses are output after the update operation is performed. CR_9 controls this mode of operation.

For all instructions accessing an offset register, the MS bit of the three-bit offset register address (B, of Bbb) is fetched from the control register and is programmed by the SELB instruction. This is also the case for the YADD and YSUB instructions (group 1) as pertains the MS bit of the four-bit address register address (R, of Rrrr), programmed by the SELR instruction. In both cases, it is incumbent upon the programmer to ensure the appropriate register bank is selected.

The Y port is only driven on output instructions (mnemonic form Yxxx, see MNEMONICS AND OPCODES). Otherwise, the Y port defaults to a high impedance state.

Instruction Group 1: Looping

Instructions in the looping group supply the contents of a selected address (R) register to the address (Y) port and then overwrite the R location with an updated value.

All instructions in this group generate an internal CMP status indicating whether the supplied address has moved to or beyond the boundary specified by the compare register. This status may be monitored externally via the CMP/Z pin. Internal to the chip, the CMP status can i) be ignored, ii) be used to control re-initialization of the R register value with a selected I register value (e.g., to restart an addressing loop), or iii) control execution of an instruction located in the AIR on the next cycle. Individual control register bits determine which option is enforced (see Control Register).

YINC Output & Increment/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + 1$.

Post-Update Mode: $Y \leftarrow R_n + 1$;
IF ($Y \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + 1$.

Output an address (R) register on the address (Y) port and compare it to one of the compare (C) registers. If the address is less than C_j, the R location is simply updated with an incremented value. However, if $R_n \geq C_j$, CMP status goes HI and the R register is re-initialized with the I_j value, provided the initialization mask (CR_{3-0}) is enabled for I_j. Note that other modes of operation allow CMP status to be ignored (e.g., the instruction executed is simply "$Y \leftarrow R_n$; $R_n \leftarrow R_n + 1$") or to cause the AIR instruction to execute on the next cycle.

YDEC Output & Decrement/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - 1$.

Post-Update Mode: $Y \leftarrow R_n - 1$;
IF ($Y \leq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - 1$.

Same as above except the R value is decremented instead of incremented; CMP is valid if the R value is less than or equal to the C value.

YADD Output & Add Offset/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + B_m$.

Post-Update Mode: $Y \leftarrow R_n + B_m$;
IF ($Y \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n + B_m$.

Same as YINC except the R value is summed with the contents of a selected offset (B) register.

The R register bank select bit (CR_7) is used in both the YADD and YSUB (offset) instructions.

YSUB Output & Subtract Offset/Init.

Pre-Update Mode: $Y \leftarrow R_n$;
IF ($R_n \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - B_m$.

Post-Update Mode: $Y \leftarrow R_n - B_m$;
IF ($Y \geq C_j$):
THEN $R_n \leftarrow I_j$,
ELSE $R_n \leftarrow R_n - B_m$.

Same as YADD except the selected offset (B) register is subtracted from the R value.

Instruction Group 2: Register Transfers

Instructions in the register transfer group support internal register transfers, as well as transfers between internal and external registers. Internally, any I or B register may be written directly to any R register. Also, any R register may simultaneously be output and written directly to a B or C register. For an R-to-R transfer, the source R register can first be written to a B register, followed by a write of the B register to an R register on the next cycle.

Internal registers are read or written externally via the bi-directional data port. There are explicit instructions to read any of these registers; however, only the I registers have an explicit Write instruction. The R, B, and C registers may be written with external data by executing a transfer instruction (YRTR, YRTB, and YRTC) and asserting the DSEL pin, substituting the external data for the designated R value.

YRTR Output & Transfer Addr. Reg. to Self

$Y \leftarrow R_n$

Outputs selected address (R) register over the address (Y) port. When DSEL is asserted, data port values are output and, in the same cycle, written into the selected R register.

YRTB Output & Transfer Addr. Reg. to Base Reg.

$Y \leftarrow R_n; B_m \leftarrow R_n$

Outputs selected R register over the Y port and copies it into a selected B register. When DSEL is asserted, data port values are output and, in the same cycle, written into the selected B register.

YRTC Output & Transfer Addr. Reg. to Comp. Reg.

$Y \leftarrow R_n; C_j \leftarrow R_n$

Same as above, except that values are written to a C register.

DTI Transfer Data Bus to Init. Reg.

$I_j \leftarrow D$

Loads selected I register from data (D) port.

ITR Transfer Init. Reg. to Addr. Reg.

$R_n \leftarrow I_j$

Selected R register is loaded from an I register, allowing a microprogram to restart a loop at any time.

BTR Transfer Base Reg. to Addr. Reg.

$R_n \leftarrow B_m$

Loads an R register from a B register. Once in the R register, the B value may be modified and then returned to the B file (using a YRTB instruction).

RTD Transfer Addr. Reg. to Data Bus

$D \leftarrow R_n$

Supplies selected R register to data (D) port.

CTD Transfer Comp. Reg. to Data Bus

$D \leftarrow C_j$

Supplies selected C register to data (D) port.

BTD Transfer Base Reg. to Data Bus

$D \leftarrow B_m$

Supplies selected B register to data (D) port.

ITD Transfer Init. Reg. to Data Bus

$D \leftarrow I_j$

Supplies selected I register to data (D) port.

Instruction Group 3: Logical & Shift

Instructions in the logical/shift group supply a value from a selected address (R) register to the address (Y) port and then unconditionally overwrite the selected R location with a modif version of the output. Modify operations include logical (AN OR, and XOR) and shift (one-bit left/right) operations. All instructions in this group affect the ZERO flag, which goes H if the result of the modification is zero. The ZERO flag statu available externally over the CMP/Z pin.

YOR Output & Logical OR to Addr. Reg.

$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ OR } B_m)$

Selected R register is supplied to the address (Y) port; the speci R location is then overwritten with the logical OR of the B register and original R value.

YAND Output & Logical AND to Addr. Reg.

$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ AND } B_m)$

Same as above, except that a logical AND is performed.

YXOR Output & Logical XOR to Addr. Reg.

$Y \leftarrow R_n; R_n \leftarrow (R_n \text{ XOR } B_m)$

Same as above, except that a logical XOR is performed.

YASR Output & Arithmetic Right Shift to Addr. Reg.

$Y \leftarrow R_n; R_n \leftarrow \text{ASR}(R_n)$

Selected R register is supplied to the address (Y) port; the speci R location is then overwritten with the original R value arithm cally shifted right (ASR) by one bit (the MSB is repeated).

YLSL Output & Logical Left Shift to Addr. Reg.

$Y \leftarrow R_n; R_n \leftarrow \text{LSL}(R_n)$

Selected R register is supplied to the address (Y) port; the speci R location is then overwritten with the original R value logic shifted left (LSL) by one bit (the LSB is zero-filled).

Instruction Group 4: Control Register

Instructions in the control register group reset, read, and wr the entire control register or individual control register bits (Control Register).

Note the use of "x" and "pp" to denote values supplied with the opcode field (see MNEMONICS AND OPCODES). A positive logic convention is used throughout.

RST Reset Control Reg.

$CR \leftarrow 0$

Clears the entire control register (CR_{10-0}). The RST instruction has dedicated decoding logic so that it takes precedence even over the second instruction of a conditional AIR sequence.

DTCR Transfer Data Bus to Control Reg.

$CR \leftarrow D$

Writes the entire control register (CR_{10-0}) from the data port, D_{10-0}.

CRTD Transfer Control Reg. to Data Bus

$D \leftarrow CR$

Outputs the entire control register (CR_{10-0}) over the data port, D_{10-0}.

SETI Set/Clear Conditional Init. on CMP Flag

$CR_{jj} \leftarrow x$

Enables conditional re-initialization of an R location, subject to CMP status (see Control Register). This instruction loads the x value into the control register bit specified by jj. Conditional re-initialization of address registers by the C_{jj}/I_{jj} pair is inhibited if the corresponding CR_{jj} is cleared.

SETP Set Chip precision

$CR_{5-4} \leftarrow pp$

Loads a 2-bit code (pp) into control register bits 5 and 4, specifying the addressing mode of the device:

00 = single-precision mode;
01 = double-precision mode, LS chip;
10 = double-precision mode, MS chip;
11 = double-precision mode, single-chip.

If the instruction "SETP, 01" is supplied and the chip's DSEL pin is asserted, the CR_{5-4} bits are reversed, i.e., the relevant chip is loaded with "10", not "01" (see Precision Modes). This is useful if the MS and LS chips share a common instruction bus.

SETY Set Y Port to Transparent/Latched Mode

$CR_6 \leftarrow x$

Uses the LS instruction bit to set the address (Y) port to the transparent (HI) or latched (LO) mode. This status is maintained in control register bit 6.

SELR Select Upper/Lower Addr. Reg. Bank

$CR_7 \leftarrow x$

The LS bit of this instruction provides the missing Address (R) register select bit required by the YADD and YSUB instructions. This selection is maintained in control register bit 7.

SELB Select Upper/Lower Base Reg. Bank

$CR_8 \leftarrow x$

The LS bit of this instruction provides the missing B register select bit required by all instructions utilizing offset (B) registers. This selection is maintained in control register bit 8.

SETU Set Update Mode (Post/Pre)

$CR_9 \leftarrow x$

Setting this bit causes the chip to output address values after updating them (post-update mode). The LS bit of this instruction determines the value of control register bit 9.

SETA Set/Clear Conditional AIR Execute Mode

$CR_{10} \leftarrow x$

Setting this bit causes Looping instructions—conditional on CMP status being HI—to execute the following instruction from the AIR on the next cycle. In this mode, conditional re-initialization of R by I on CMP is inhibited. The LS bit of this instruction determines the value of control register bit 10.

Instruction Group 5: AIR Control

Instructions in the AIR group write and read the Alternate Instruction Register (AIR). The AIR may be written or read over the data bus in one cycle or written via the instruction port in two cycles (see Table I). The instruction contained in the AIR is executed whenever the AIR Enable pin is asserted or on the next cycle in the conditional AIR execute mode.

WRA Write AIR with Data Bus

$AIR \leftarrow D$

Write the AIR from the data (D) bus (D_{9-0}).

RDA Read AIR at Data Bus

$D \leftarrow AIR$

Read the AIR over the data (D) bus (D_{9-0}).

LDA Load AIR from Instruction Port on Next Cycle (Requires DSEL HI during load)

$AIR \leftarrow$ Instruction Port

This instruction is the first of a two-cycle sequence that loads the AIR via the instruction port. On the cycle following its execution, the instruction appearing on the instruction port will be loaded into the AIR—provided that the DSEL pin is asserted. If DSEL is not asserted during the cycle following LDA, the AIR is not loaded and a NOP is executed, superseding the instruction at the instruction port.

Instruction Group 6: Miscellaneous

DTY Pass Data Bus to Y Port

$Y \leftarrow D$

Data (D) port values are supplied directly to the address (Y) port. Note that internal address (R) registers are not affected by this instruction.

YREV Output Addr. Reg. in Bit-Reversed Format

$Y \leftarrow YREV(R_n)$; $R_n \leftarrow R_n + B_m$

The selected address (R) register is bit reversed at the output port. The original (unreversed) R value is added to the selected offset (B) register, and written back into the specified R location. Condition testing is not performed. Bit reversing affects only output data, not register contents.

NOP No Operation

Prevents any changes to the internal conditions of the AG. All I/O pins go to the three-state disable mode.

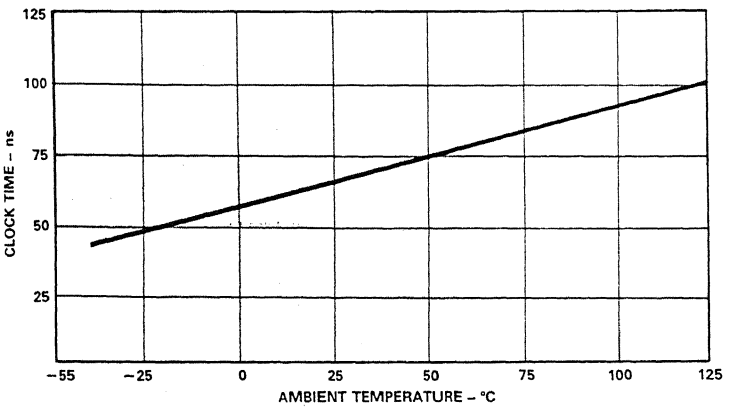


Figure 7. Clock Cycle Time vs. Temperature

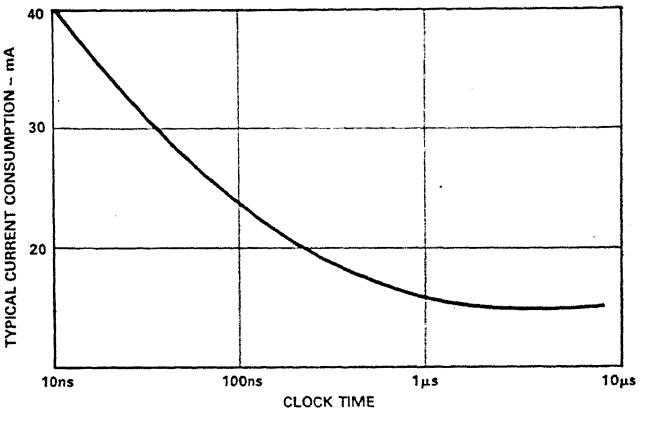


Figure 8. Typical I_{DD} vs. Frequency of Operation

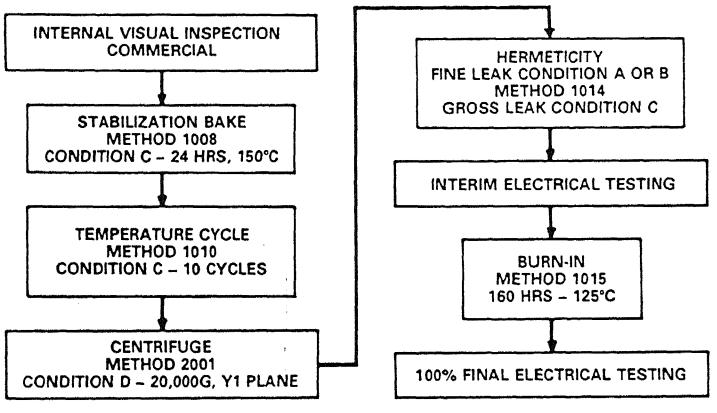


Figure 9. PLUS Processing Environmental Flow

MNEMONICS AND OPCODES

The following list gives the instruction mnemonics and opcodes. Various parameters are substituted by the user, defining register numbers or control bits. The notation convention is this:

R	=	Address register
B	=	Base (offset) register
C	=	Compare register
I	=	Initialization register
D	=	Data bus
CR	=	Control register
rrrr	=	Four-bit address register number
rrr	=	Three-bit address register number
bb	=	Two-bit base (offset) register number
cc	=	Two-bit comparison register number
ii	=	Two-bit initialization register number
pp	=	Two-bit precision code
x	=	One-bit control bit

*External data may substitute for R using DSEL.
†Operable in either pre- or post-update mode.

Instr.	Opcode (I_{9-0})	Description
Looping Instructions*†		
YINC:	1011ccrrrr	output & increment/init
YDEC:	1010ccrrrr	output & decrement/init
YADD:	11ccbb1rrr	output & add offset/init
YSUB:	11ccbb0rrr	output & subtract offset/init
Register Transfer Instructions		
YRTR*:	000101rrrr	output & xfr R to R
YRTB*:	0011bbrrrr	output & xfr R to B
YRTC*:	0010ccrrrr	output & xfr R to C
DTI:	00001111ii	xfr D to I
ITR:	1000iirrrr	xfr I to R
BTR:	0100bbrrrr	xfr B to R
RTD:	000100rrrr	xfr R to D
CTD:	00001100cc	xfr C to D
BTD:	00001101bb	xfr B to D
ITD:	00001110ii	xfr I to D
Logical and Shift Instructions*†		
YOR:	0111bbrrrr	output & OR B with/to R
YAND:	0110bbrrrr	output & AND B with/to R
YXOR:	0101bbrrrr	output & XOR B with/to R
YASR:	000111rrrr	output & arith SR R to R
YLSL:	000110rrrr	output & logical SL R to R
Control Register Instructions		
RST:	0000000001	reset CR
DTCR:	0000101110	xfr D to CR
CRTD:	0000101111	xfr CR to D
SETI:	0000100iix	set cond re-init on CMP mode
SETP:	00001010pp	set chip precision
SETY:	000001001x	set Y port to trans/latched mo
SELR:	000001101x	select upper/lower R bank
SELB:	000001100x	select upper/lower B bank
SETU:	000001011x	set post/pre update mode
SETA:	000001010x	set cond AIR mode
AIR Instructions		
WRA:	0000101100	write AIR with D
RDA:	0000101101	read AIR at D
LDA:	0000011110	load AIR on next cycle
Misc. Instructions		
YDTY:	0000011111	pass D to Y port
YREV*†:	1001bbrrrr	output R in bit-reverse form
NOP:	0000000000	no operation

ANALOG DEVICES

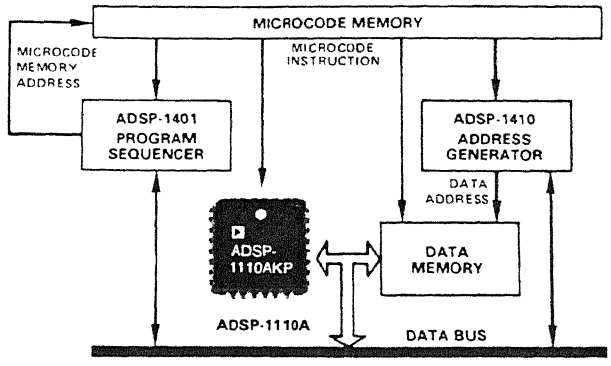
16 × 16-Bit CMOS Single Port Multiplier/Accumulator

FEATURES

16 × 16-Bit Parallel Multiplication/Accumulation
40-Bit Wide Accumulator with Overflow Flag, Saturation Arithmetic, and Shift-Left Control
Twos Complement or Unsigned Magnitude Inputs
85ns Multiply/Accumulate Time
28-Lead Ceramic DIP, Plastic DIP Package, Plastic Leaded Chip Carrier, or Leadless Chip Carrier
350mW Power Dissipation with CMOS Technology
Specified Over the Extended Temperature Range
Pin-Compatible with ADSP-1110

APPLICATIONS

Digital Filtering
Fast Fourier Transforms
Matrix Multiplication
Microprocessor Acceleration



WORD-SLICE™ MICROCODED SYSTEM WITH ADSP-1110A

GENERAL INFORMATION

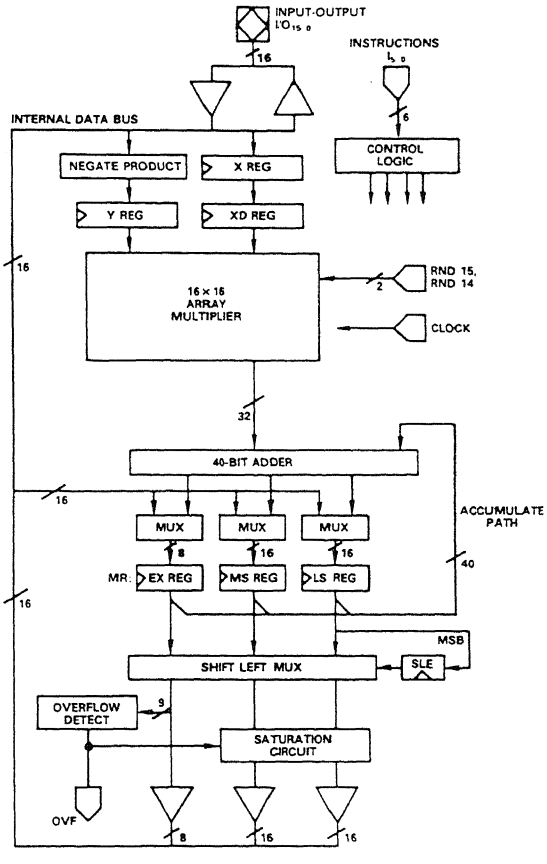
The ADSP-1110A is a high-speed, low-power single-port 16 × 16-bit multiplier/accumulator (MAC), with processing throughput comparable to existing three-port MACs. Its single-bus structure offers unique advantages: more compact packaging in a 28-pin package, simpler system interface to single-bus peripherals, and significantly reduced cost. In addition, innovative on-chip features extend the ADSP-1110A's capabilities and eliminate external hardware.

All inputs to and outputs from the ADSP-1110A pass through its single 16-bit I/O port. All I/O operations are single cycle. A multiplication or MAC operation requires two cycles to complete—consistent with the two cycles required to load input pairs to the multiplier. An internal pipeline register enables a new input to be loaded as the previous multiplication/accumulation is computed—allowing the device's full 11.7MHz computational bandwidth to be utilized.

A six-bit microcode instruction word governs the ADSP-1110A's operation. The instruction set centers around I/O and multiplication/accumulation operations. Additional instructions allow extra precision in single- and double-precision operations to be obtained efficiently.

Multiplier products are accumulated in a 40-bit wide Multiplier Result (MR) register, which consists of a 16-bit MS (Most Significant) and LS (Least Significant) register, and an 8-bit EX (Extension) register. Either multiplier input can be a twos complement or unsigned magnitude number. Overflow from the lower 32 bits of the MR into the upper eight guard bits is detected and can be monitored externally. Outputs can, conditional upon overflow status, be saturated to full scale. An MR register can be shifted left by one bit upon output; two independent controls allow rounding consistent with output formatting.

The ADSP-1110A is optimal for applications where board space is limited but the performance of a DSP processor is required. In addition, a microprocessor-based system can realize greater throughput by utilizing the ADSP-1110A in an accelerator.



ADSP-1110A Functional Block Diagram

One Technology Way; P. O. Box 9106; Norwood, MA 02062-9106 U.S.A.
Tel: 617/329-4700 Twx: 710/394-6577
Telex: 174059 Cables: ANALOG NORWOODMASS

METHOD OF OPERATION

The ADSP-1110A's operation is controlled by a six-bit microcode instruction and two rounding control pins. Table III presents instructions that are executed by the ADSP-1110A, along with the corresponding six-bit microcode instruction. The sections below further describe the instruction groups presented in Table III.

Input and Multi–Operation Instructions

A dedicated *input instruction* ("X = Bus") loads the X input at the rising edge of the clock. The X input is loaded with the data that is set up on the device's 16-bit I/O port.

A set of *multi-operation instructions* ("Y = BUS; CKMR; X*Y") are used to load the Y input and otherwise control the ADSP-1110A's multiplier/accumulator. Specifically, at the next rising clock edge, a multi-operation instruction i) loads Y input ii); clocks the result of the previous multiplication/MAC operation into the MR; and, iii) initiates the next multiplication/MAC operation. The multiplication/MAC operation is initiated at the rising edge of the clock and requires two cycles to complete. The instruction controls needed to govern the device's multiplier array and 40-bit adder during these two cycles are registered internally.

During the first cycle of a multi-operation instruction, the X input is transferred to an internal pipeline register (XD), and is latched there on the next rising clock edge. Consequently, a new X value can be loaded onto the chip during the second cycle of the multi-operation instruction. XD will not be overwritten until a new X value is loaded.

The ADSP-1110A supports the following multiplication and multiplication/accumulation operations:

$$\pm X*Y$$

and,

$$\pm X*Y \pm MR$$

The ADSP-1110A allows either input to be specified as a twos complement or unsigned magnitude number. Table II describes, for all combinations of inputs, the proper interpretation of the MR register if it is output with or without the left-shift option. Note that if the Y input is negative full scale and a negative product is specified, an invalid result is obtained. This happens because the ADSP-1110A will attempt to produce the unrepresentable twos complement of full-scale negative.

The result of a multiplication or MAC operation is latched into the MR register in either of two ways. A dedicated "CKMR" instruction performs this clocking. In addition, all multi-operation instructions clock the MR, eliminating overhead when computing MAC's (see *Instruction Sequences*). It is important to note that whenever "CKMR" is executed, it clocks the result of the *previous* operation into the MR. Also, in all cases, the clocking of the MR occurs at the rising edge of the clock.

MR Register Instructions

A number of the ADSP-1110A's instructions affect the contents of the MR register—including *preload instructions*, *transfer instructions*, and *sign extend instructions*. In addition, special output instructions allow for format adjusting the MR upon output.

The 40-bit accumulator of the ADSP-1110A is segmented into three registers: a 16-bit most significant product register (MS); a 16-bit least significant product register (LS); and, an 8-bit extended product register (EX) (see Table II). The eight guard bits of the EX allow at least 256 multiplication/accumulations without risk of overflow.

Dedicated instructions allow any of the MR's registers to be preloaded with data set up on the device's 16-bit I/O port. This preloading occurs at the rising edge of the clock.

The proper sequence for preloading a value Z into MR and adding it to the product $X_1 * Y_1$ is:

Instruction	Comment
1. X = BUS	Load X_1
2. Y = BUS; CKMR; X*Y + MR	Load Y_1; clock garbage into MR; initiate MAC
3. LS = BUS	Preload MR with Z
4. MS = BUS	Preload MR with Z
5. EX = BUS	Preload MR with Z
6. X = BUS	Load X_2
7. Y = BUS; CKMR; X*Y + MR	Load Y_2; MR = $X_1*Y_1 + Z$; initiate next multiplication.

This sequence ensures that the value Z preloaded by instructions 3, 4, and 5 is added to the product X_1*Y_1 and clocked into MR by instruction 7. If Z were preloaded prior to instruction 2, then instruction 2's "CKMR" operation would overwrite the Z value with the product of whatever values were last placed in the multiplier array.

Transfer operations allow one MR register to be moved down to an adjacent one—useful in double-precision operations. The ADSP-1110A can, in one cycle, shift the EX to the MS or the MS to the LS register. The shift left extend register (SLE) is a one-bit latch that is loaded with the value of the MSB of the LS register whenever the MS is transferred to LS. The SLE register retains its value until the next downshift of MS into LS overwrites its contents.

Anytime the result of a multiplication or multiplication/accumulation operation is clocked into the accumulator, the result is automatically sign extended into the upper MSBs of the accumulator. In addition, explicit instructions allow the MSB of the LS to be sign extended to the MS ("MS = SIGN EXT LS") or the MSB of the MS to be sign extended to the EX register ("EX = SIGN EXT MS"). Such sign extend capability may be needed to properly initialize the MR after the MS or LS is preloaded, or after an MR register transfer.

Output Instructions

Output instructions allow any MR register to be read. When written onto the ADSP-1110A's 16-bit bus, the 8-bit EX register is automatically sign-extended into the upper 8 MSBs of the bus. Standard output instructions of the ADSP-1110A are supplemented with two important options: a shift-left capability and conditional saturation.

The ADSP-1110A's output instructions include the ability to shift any MR register (EX, MS, or LS) left by one bit upon output. This shift does not affect the contents of MR, but does affect what appears on the ADSP-1110A's 16-bit I/O port. Figure 6 shows which bits of the 40-bit wide MR register are output if the shift-left option is invoked.

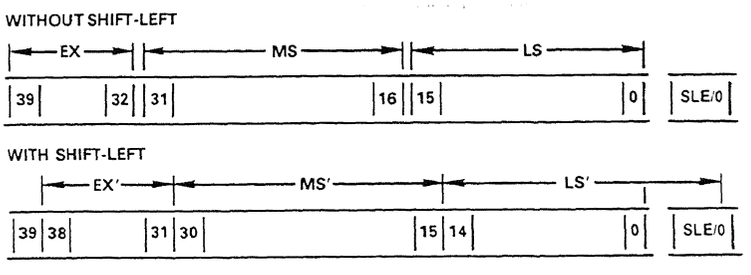


Figure 6. Effect of Left Shift on MR Outputs

The shift left-on-output control, which scales up the MR outputs by a factor of two, is useful under many circumstances. Twos complement multiplication—for all but one case (negative full-scale times negative full-scale)—results in redundancy in the two MSBs of the 32-bit product. This redundancy means that the 16-bit MS register contains two identical sign bits (bits 31 and 30 of MR) and just 14 bits of magnitude. The ADSP-1110A's shift-left control allows full precision in twos complement operations to be attained. Left shift control also provides a means for maximizing resolution when using block floating point, when downscaling twos complement results, and when upscaling mixed and unsigned magnitude results.

Whenever the RND14 pin is asserted during a "BUS = LS (sl)" or "BUS = LS (sl, sat)" instruction, the SLE bit will be appended to the upper 15 bits of the shifted LS. If the RND14 is low, however, a zero will be inserted into the LSB of LS. Appending the SLE bit to the shifted LS provides an extra bit of precision in applications such as double-precision multiplication/accumulations.

Round Controls

The RND14 and RND15 pins are two independent controls that allow rounding consistent with shifted or unshifted outputs, respectively. The round control signals are latched at the rising clock edge whenever the device receives a multiplication or MAC instruction. Asserting the RND 15 (RND14) pin will cause a 1 to be added to bit 15 (bit 14) of the LS. The rounding will not occur until the subsequent cycle in which the result of the multiplication or MAC operation is clocked into the MR.

Overflow and Saturation

The ADSP-1110A's overflow flag monitors 9 bits (8 in the EX register and the MSB of the MS register). If any bits in EX differ from the MSB of MS, then an overflow has occurred from the MS into the EX, and the overflow flag is asserted (HI) following an output instruction. Generally, the status of the overflow flag reflects the current contents of the MR and is updated each time a new result is clocked into the MR. However, if the MS register is output with a left-shift, the overflow logic determines whether the shifted MS overflows into the EX (when bits 38 through 30 are not identical) and is set accordingly. On the cycle following any left-shifted output, the overflow flag status reverts to reflect the contents of the MR. During cycles when a non-output instruction is executed, the overflow flag is always LO

Serious data glitches can result from wraparound effects due to overflow in long multiply/accumulate chains. For example, if a positive number is added to positive full-scale, the 32 MSBs of the MR register will overflow into the 8-bit EX register. Simply reading the MS register will yield a negative twos complement number. To prevent this wraparound, the ADSP-1110A can—conditioned on overflow status—saturate an output to twos-complement full scale.

The ADSP-1110A's saturation logic operates only on output values; it has no effect on the contents of the MR register. This logic examines the sign of the MR (bit 39, the MSB of the EX register) and the overflow status. As Table IV indicates, the low 32 bits of the MR are saturated to full-scale positive (negative) if overflow has occurred in a positive (negative) MR.

Either the MS or LS registers can be left-shifted on output with conditional saturation. If the shifted value overflows the lower 32 bits, the outputted result will be saturated to full scale.

While the saturation control protects against overflow from the MS to the EX register, the user is not protected in the event the accumulated result overflows the entire 40-bit MR register.

OVF	MR BIT 39 (SIGN BIT)	Output Value with Saturation MS	LS
0	0	← No Change →	
0	1	← No Change →	
1	0	0111111111111111	1111111111111111
1	1	1000000000000000	0000000000000000

Table I. Overflow and Saturation Circuitry Conditions

INPUTS (X & Y)				EX (8 BITS)				MR MS (16 BITS)					LS (16 BITS)			
b_{15}	b_{14}	------	b_0	b_{39}	b_{38}	------	b_{32}	b_{31}	b_{30}	------	b_{16}	b_{15}	------	b_1	b_0	
TWOS COMPLEMENT INTEGER				(w/o sl)												
-2^{15}	2^{14}	------	2^0	-2^{39}	2^{38}	------	2^{32}	2^{31}		------	2^{16}	2^{15}	------	2^1	2^0	
				(w/ sl)												
				-2^{38}	2^{37}	------	2^{31}	2^{30}		------	2^{15}	2^{14}	------	2^0	SLE/0	
TWOS COMPLEMENT FRACTIONAL				(w/o sl)												
-2^0	2^{-1}	------	2^{-15}	-2^9	2^8	------	2^2	2^1	2^0	------	2^{-14}	2^{-15}	------	2^{-29}	2^{-30}	
				(w/ sl)												
				-2^8	2^7	------	2^1	2^0	2^{-1}	------	2^{-15}	2^{-16}	------	2^{-30}	SLE/0	
UNSIGNED MAGNITUDE INTEGER																
2^{15}		------	2^0	2^{39}		------	2^{32}	2^{31}		------	2^{16}	2^{15}	------	2^1	2^0	
UNSIGNED MAGNITUDE FRACTIONAL																
2^{-1}		------	2^{-16}	2^7		------	2^0	2^{-1}		------	2^{-16}	2^{-17}	------	2^{-31}	2^{-32}	
MIXED MODE INTEGER																
-2^{15}	2^{14}	------	2^0 &													
2^{15}	2^{14}	------	2^0	-2^{39}	2^{38}	------	2^{32}	2^{31}		------	2^{16}	2^{15}	------	2^1	2^0	
MIXED MODE FRACTIONAL																
-2^0	2^{-1}	------	2^{-15} &													
2^{-1}	2^{-2}	------	2^{-16}	-2^8	2^7	------	2^1	2^0		------	2^{-15}	2^{-16}	------	2^{-30}	2^{-31}	

Instruction Group	Instruction	Microcode Instruction 5 4 3 2 1 0	Comments
Miscellaneous	NOP	0 0 0 0 x x	No Operation
	CKMR	0 0 0 1 x x	Clock MR
Input	X = BUS	0 0 1 0 x x	
Preload	LS = BUS	0 1 0 0 0 0	
	MS = BUS	0 1 0 1 x 0	
	EX = BUS	0 1 0 0 1 0	
Transfer	LS = MS	0 1 0 0 0 1	Sets SLE register
	MS = EX	0 1 0 1 0 1	
Sign Extend	EX = SIGN EXT MS	0 1 0 0 1 1	
	MS = SIGN EXT LS	0 1 0 1 1 1	
Output	BUS = EX	0 0 1 1 0 1	All output instructions are asynchronous
	BUS = EX (sl)	0 0 1 1 0 0	I5 – I2:
	BUS = MS	0 1 1 0 0 1	0011 = EX
	BUS = MS (sl)	0 1 1 0 0 0	0110 = MS
	BUS = MS (sat)	0 1 1 0 1 1	0111 = LS
	BUS = MS (sl,sat)	0 1 1 0 1 0	I1 – I0:
	BUS = LS	0 1 1 1 0 1	01 = to bus
	BUS = LS (sl)	0 1 1 1 0 0	00 = to bus shifted
	BUS = LS (sat)	0 1 1 1 1 1	10 = to bus shifted w/saturation
	BUS = LS (sl,sat)	0 1 1 1 1 0	11 = to bus w/saturation
Multi-Operation	Y = BUS; CKMR; $X_{US}*Y_{US}$	1 0 0 x 0 0	Require two cycles to complete.
	Y = BUS; CKMR; $-X_{US}*Y_{US}$	1 0 0 x 0 1	Other instructions can be executed
	Y = BUS; CKMR; $X_{US}*Y_{US}$ + MR	1 0 0 0 1 0	on the second cycle.
	Y = BUS; CKMR, $-X_{US}*Y_{US}$ + MR	1 0 0 0 1 1	
	Y = BUS; CKMR; $X_{US}*Y_{US}$ − MR	1 0 0 1 1 0	I5 = Multiplication/MAC operation
	Y = BUS; CKMR; $-X_{US}*Y_{US}$ − MR	1 0 0 1 1 1	I4 = Y twos complement
	Y = BUS; CKMR; $X_{TC}*Y_{US}$	1 0 1 x 0 0	I3 = X twos complement
	Y = BUS; CKMR; $-X_{TC}*Y_{US}$	1 0 1 x 0 1	I2 = Subtract previous result
	Y = BUS; CKMR; $X_{TC}*Y_{US}$ + MR	1 0 1 0 1 0	I1 = Add/subtract previous result
	Y = BUS; CKMR; $-X_{TC}*Y_{US}$ + MR	1 0 1 0 1 1	from product
	Y = BUS; CKMR; $X_{TC}*Y_{US}$ − MR	1 0 1 1 1 0	I0 = Negate product
	Y = BUS; CKMR; $-X_{TC}*Y_{US}$ − MR	1 0 1 1 1 1	
	Y = BUS; CKMR; $X_{US}*Y_{TC}$	1 1 0 x 0 0	
	Y = BUS; CKMR; $-X_{US}*Y_{TC}$	1 1 0 x 0 1	
	Y = BUS; CKMR; $X_{US}*Y_{TC}$ + MR	1 1 0 0 1 0	
	Y = BUS; CKMR; $-X_{US}*Y_{TC}$ + MR	1 1 0 0 1 1	
	Y = BUS; CKMR; $X_{US}*Y_{TC}$ − MR	1 1 0 1 1 0	
	Y = BUS; CKMR; $-X_{US}*Y_{TC}$ − MR	1 1 0 1 1 1	
	Y = BUS; CKMR; $X_{TC}*Y_{TC}$	1 1 1 x 0 0	
	Y = BUS; CKMR; $-X_{TC}*Y_{TC}$	1 1 1 x 0 1	
	Y = BUS; CKMR; $X_{TC}*Y_{TC}$ + MR	1 1 1 0 1 0	
	Y = BUS; CKMR; $-X_{TC}*Y_{TC}$ + MR	1 1 1 0 1 1	
	Y = BUS; CKMR; $X_{TC}*Y_{TC}$ − MR	1 1 1 1 1 0	
	Y = BUS; CKMR; $-X_{TC}*Y_{TC}$ − MR	1 1 1 1 1 1	

Mnemonic Definitions

=	Assign right side to left.	sl	Shift left.
BUS	16-bit external data bus used for all I/O operations.	sat	Conditional on overflow, saturate the outputted value.
X	Input register for multiplier.	TC	Two's complement number.
Y	Input register for multiplier.	US	Unsigned magnitude number.
EX	8-bit extension register for accumulator.	SIGN	Sign bit (MSB) of specified register.
MS	16-bit most significant product register.	CKMR	Clock product into EX, MS, and LS.
LS	16-bit least significant product register.	*	Multiply
MR	40-bit accumulator comprising EX, MS and LS.	x	Microcode instruction bit can be either a 0 or 1.

Table III. ADSP-1110A Instruction Set

CLOCK AND TIMING

Figure 1 presents a timing diagram for the ADSP-1110A's operation.

Input data, round controls, and non-output instructions are clocked (synchronous); set-up and hold times are specified accordingly. All multi-operation (two cycle) instructions are clocked, and the internal controls needed for the second cycle are latched internally.

Unlike all other ADSP-1110A instructions, output operations are asynchronous. The relevant timing specification is the delay between control inputs and valid outputs. The use of saturation (sat) slows down the availability of a valid output on the ADSP-1110A's I/O bus; delay times are specified accordingly.

The ADSP-1110A's OVF (overflow) flag is set according to the contents of the MR register. However, upon outputting the MR with the shift-left control, the OVF flag may be modified if the left shift causes overflow. The relevant timing for this case is specified.

The ADSP-1110A's output three-state drivers are not disabled until t_{DIS} ns after an output instruction is removed. Since the ADSP-1110A has just one I/O port, bus contention can occur when an ADSP-1110A input immediately follows an output. For example, an input source (e.g., a data RAM) enabled to drive the bus immediately after an ADSP-1110A output creates the possibility that both drivers are active simultaneously. There are two ways to avoid such conflicts:

1. Set up output instructions well in advance of the clock's rising edge (>t_D set-up time), enabling the data output to complete in time for the data to be latched at the clock edge. Allow t_{DIS} ns after the clock edge before enabling a different device to drive the bus. Note that any system that provides the ADSP-1110A with its instruction from a pipeline register operates in this way. The *Hardware Implementations with the ADSP-1110A* section describes several alternative implementations consistent with this approach.
2. For systems with minimal instruction set-up time, an operation that doesn't use the bus (e.g., a NOP) may need to be inserted after an output instruction. The reason for this is as follows. Output instructions must be held valid for t_D ns, which means that—if instruction set-up time is minimal—output instructions must be held beyond the rising edge of the clock. After the output instruction is removed, another t_{DIS} elapses before the output drivers are disabled. As a result, the three-state output drivers are active well into the next cycle. If the bus is driven with an input in the next cycle, bus contention may occur.

Instruction Sequences

With the ADSP-1110A, single multiplication operations involve three overhead statements in addition to the multiply command, as Figure 7 illustrates.

While a multiplication/accumulation sequence is structurally similar to a single multiplication, overhead as a percentage of computation time is reduced substantially. In the instruction flow diagram shown in Figure 8, a NOP is needed only in the final multiplication/accumulation operation. Also, new X values are loaded as multiplication/accumulation instructions complete. In this sequence, the three cycles of overhead can be spread out over as many multiplication/accumulations as are performed consecutively.

For a series of multiplication/accumulation sequences, I/O operations can be further overlapped. At the end of each multiplication/accumulation string, a new string is initiated. In this instance, overhead cycles become negligible in importance; the multiplication/accumulation rate of the ADSP-1110A approaches 11MHz.

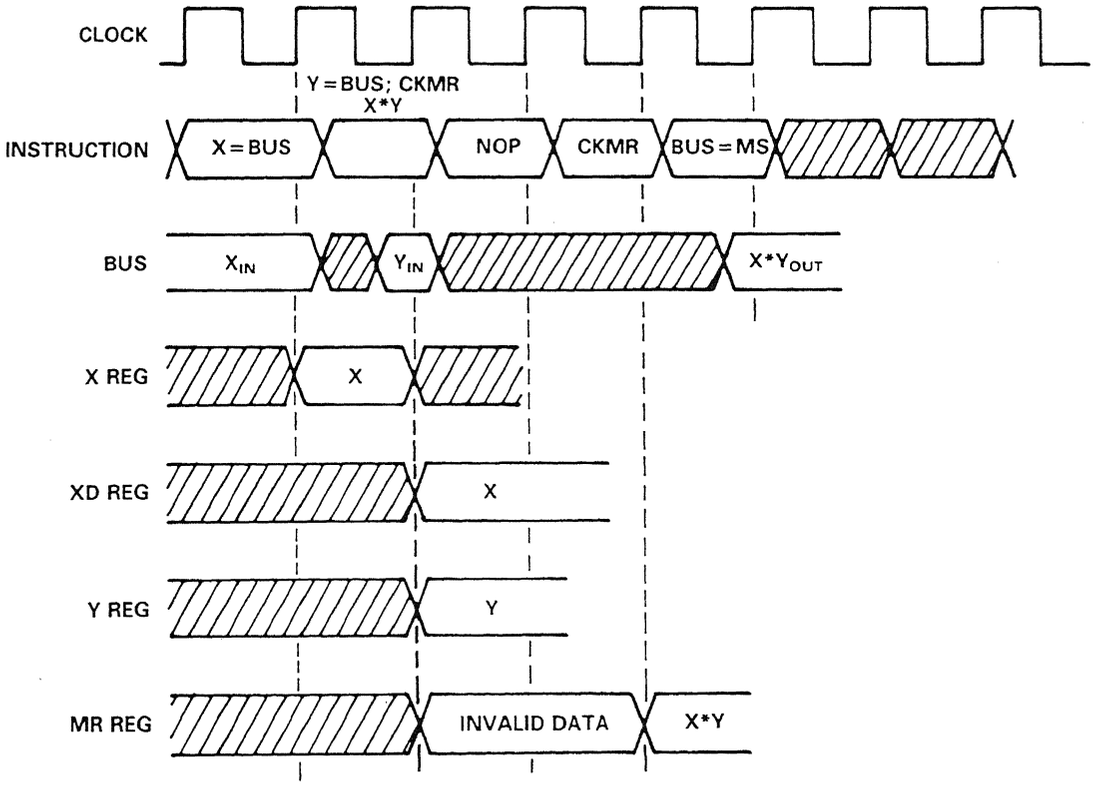


Figure 7. Multiply Operation Timing

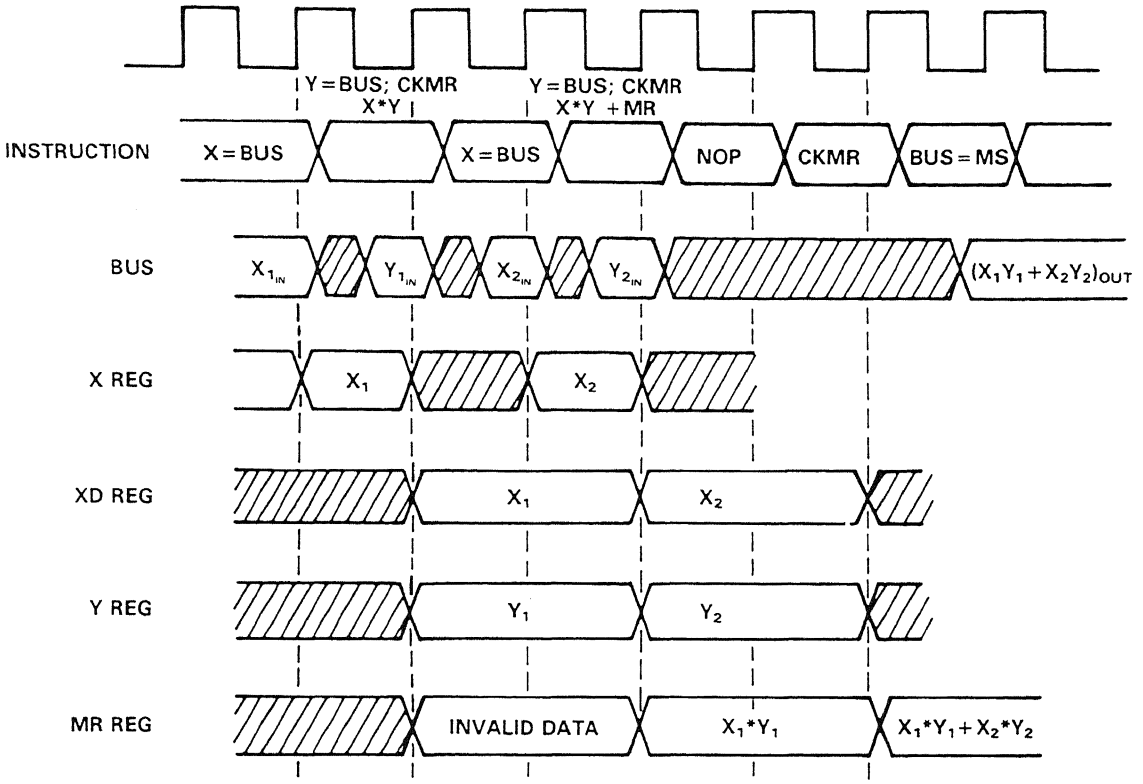


Figure 8. Multiply/Accumulate Operation Timing

Avoid Bus Contentions

Because the ADSP-1110A typically shares its data port with other devices on a common bus, there is a potential for bus contentions at power-up. If the instruction applied to the ADSP-1110A at power-up is random, the multiplier/accumulator could be in an output state. If any other devices are driving the bus at the same time, there will be a bus contention.

The obvious solution is to make sure no other devices are driving the common data bus at power-up. Another approach is to force instruction bit I_5 (pin 18) HI at power-up. This guarantees that the ADSP-1110A will not be in an output state because ADSP-1110A output instructions are asynchronous and all have a zero in instruction bit 5 (I_5).

HARDWARE IMPLEMENTATONS WITH THE ADSP-1110A

There are many alternative ways of implementing high performance DSP systems with the ADSP-1110A. The following sections illustrate some of the more commonly used approaches using the ADSP-1110A: a microcoded system, a ROM-based sequential machine, a PLA-based state machine, and as a device directly interfaced to a microprocessor. The optimal implementation will depend on the performance, price, and board area requirements of the design.

Microcoded System

Many microcoded systems have the design objective of fast number crunching, while minimizing microcode bits and circuit board area. The ADSP-1110A single port MAC—with just 8 control bits, its single bus structure, and fast cycle time—helps meet these objectives. The ADSP-1110A can be simply connected to the processor data bus and microcode instruction field to provide powerful multiplier/accumulator functions.

A typical Word-Slice™ processor with the ADSP-1110A is shown below:

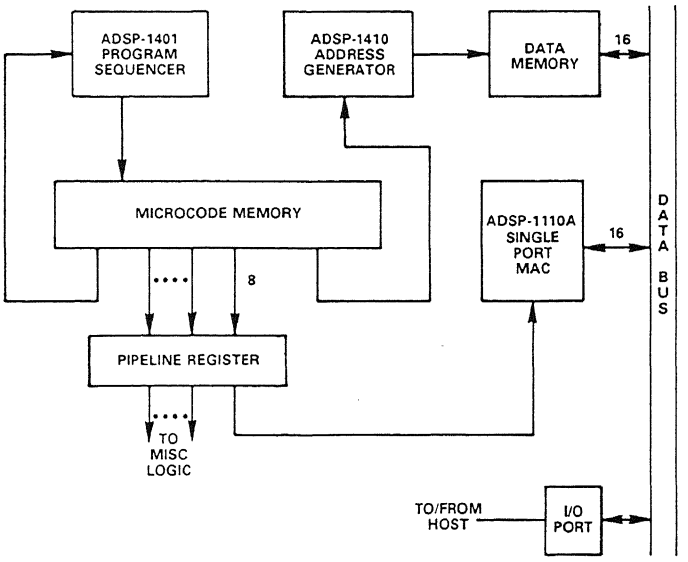


Figure 9.

In most bit-slice designs, the control bits from the microcode memory are latched in a pipeline register. In the above implementation, the ADSP-1110A and all miscellaneous logic are used in conjunction with an external pipeline latch. The pipeline latch guarantees that the microcode bits controlling the circuitry are valid for a complete cycle (see timing diagram below). Note that the ADSP Word-Slice™ components (the ADSP-1401 and ADSP-1410) contain an internal pipeline register and are fed directly from the microcode.

APPENDIX D

```
; THIS DEF. FILE IS FOR THE MICROCODED SYSTEM 1
TITLE MICROSYSTEM
DEFN_FILE "SMACRO.DAT"
WORD WIDTH 49 BITS
DATA_LOC_WIDTH 16
PROGRAM_LOC_WIDTH 49
;1
#sequencer:FIELD 42-48,WIDTH 7, DEFAULT cont
  OPCODE_BIT_NOTATIONS [
   kk (    ;Conditions
  00  unconditional
  01  notflag
  10  flag
  11  sign
  )
    cc (    ;Selects the relevent register(R3-R0)and
  C0-C3   ;/or counter(C3-C0).
  )
    ii (    ;Decides number to be added incase of
  I0-I3   ; AIRSP instruction.
  )
  ]
   INSTR_NOT_AVAILABLE [
  jrc(sign)(c0)
  jrc(sign)(c1)
  jrc(sign)(c2)
  jrc(sign)(c3)
  branch(sign)(c0)
  branch(sign)(c1)
  branch(sign)(c2)
  branch(sign)(c3)
  ]
```

```
  BRANCH_INSTR_WHICH_NEED_DATA [
ABSOLUTE (
   jsa
   jda
   jdrst
 )
RELATIVE (
 jsr
 jdr
 )
 ]
  VALUES [
;jump & branch instruction

 jpcof 0010101
 jpcnf 0110101
 jtwo  101kk01
 jda  111kk11
 jdr  111kk01
 jdi  101kk10
 jdrst 10011cc
 jrs  11011cc
 jsa  111kk00
 jsr  111kk10
 rtn  101kk11
 branch 100kkcc

 ;stack operation
 ;subroutine stack

 psdss 0011110
 ppssd 0111110
 wrssp 0001110
 rdssp 0101100
 dssp  0000010

 ;register stack
 sgsp  0000111
```

```
slsp  0000110
rdrsp 0101111
wrrsp 0001100
pspc  0100011
psgsp 0000101
ppgsp 0000100
psdrs 0011111
pprsd 0111111
airsp 01010ii
sirsp 0001111
s4rsp 0111100

;status register operations:
rdsr  0101110
wrsr   0011100
pssr  0100001
ppsr  0100010

;counter operations:
wrcntr 01110cc
clrs  0010100
sets  0110100
pscntr 00010cc
ppcntr 00110cc
dccntr 01100cc
ifcdec 101kk00

;Interrupt control:
ccir  0010001
cair  0000001
rtnir 0000011
rdiv  0101101
wriv  0001101
irmbc 0010011
irmbs 0010010
disir 0010110
enair 0110110
slir  0010111
stir  0110111
```

```
  slrivp 0011101

  ;relative address width controls:
  rel16 0100100
  rel12 0100111
  rel8  0100110

  ;miscellaneous instructions:
  cont  0000000
  idle  0010000
  ihc   0100101
  wcs   0100000


  ]
;2
#data:FIELD 26-41,WIDTH 16,DEFAULT zero

  VALUES [
 zero  0000000000000000
 ]

;3
#DATA_ENABLE:FIELD 25-25,WIDTH 1,DEFAULT disable
        VALUES [
      enable   1
      disable  0
        ]
;4
#address_generator: FIELD 15-24,WIDTH 10,DEFAULT nop

  OPCODE_BIT_NOTATIONS [
   cc (  ;Comparison Register number
  c0-c3
  )
   rrr (  ;Three bit Address register number
  R0-R7
  )
   rrrr ( ;Four bit Address register number
```

```
r0-r15
)
 bb (  ;Base (offset) register number
b0-b3
)
 ii (  ;Initialisation register number
i0-i3
)
 pp (  ;Two bit precision code
p0-p3
)
 x (  ;One bit control bit
x0-x1
)

]
VALUES [
 ;looping instructions

yinc    1011ccrrrr
ydec  1010ccrrrr
yadd  11ccbb1rrr
ysub  11ccbb0rrr

 ;register transfer instructions:
yrtr  000101rrrr
yrtb  0011bbrrrr
yrtc  0010ccrrrr
dti  00001111ii
itr  1000iirrrr
btr  0100bbrrrr
rtd  000100rrrr
ctd  00001100cc
btd  00001101bb
itd  00001110ii

 ;logical and shift instructions
yor  0111bbrrrr
yand  0110bbrrrr
```

```
 r0-r15
 )
  bb (  ;Base (offset) register number
 b0-b3
 )
  ii (  ;Initialisation register number
 i0-i3
 )
  pp (  ;Two bit precision code
 p0-p3
 )
  x (  ;One bit control bit
 x0-x1
 )

 ]
VALUES [
 ;looping instructions

yinc    1011ccrrrr
ydec  1010ccrrrr
yadd  11ccbb1rrr
ysub  11ccbb0rrr

 ;register transfer instructions:
yrtr  000101rrrr
yrtb  0011bbrrrr
yrtc  0010ccrrrr
dti  0000111iii
itr  1000iirrrr
btr  0100bbrrrr
rtd  000100rrrr
ctd  00001100cc
btd  00001101bb
itd  00001110ii

 ;logical and shift instructions
```

```
 yxor  0101bbrrrr
 yasr  000111rrrr
 ylsl  000110rrrr

 rst  0000000001
 dtcr  0000101110
 crtd  0000101111
 seti  0000100iix
 setp  00001010pp
 sety  000001001x
 selr  000001101x
 selb  000001100x
 setu  000001011x
 seta  000001010x

 wra  0000101100
 rda  0000101101
 lda  0000011110

 ydty  0000011111
 yrev  1001bbrrrr
 nop     0000000000
 ]

;5
#ag_gen_pin: FIELD 14-14,WIDTH 1,DEFAULT nodsel

  VALUES [
 nodsel 0
 dsel 1
 ]
;6
#ag_air_select: FIELD 13-13,width 1,default noair
VALUES [
 air 1
 noair 0
 ]
```

```
  VALUES
 [
 rd    11
 wr    10
 norw  00
       ]
;8
#mac:FIELD 5-10,WIDTH 6,default nop
  VALUES [
 nop          000000
 ckmr         000100
 x=bus        001000
 ls=bus       010000
 ms=bus       010100
 ex=bus       010010
 ls=ms        010001
 ms=ex        010101
 ex=signextms    010011
 ms=signextls    010111
 bus=ex       001101
 bus=ex(sl)   001100
 bus=ms       011001
 bus=ms(sl)   011000
 bus=ms(sat)  011011
 bus=ms(slsat)   011010
 bus=ls       011101
 bus=ls(sl)   011100
 bus=ls(sat)  011111
 bus=ls(slsat)   011110
 y=bus_ckmr_xus*yus          100000
 y=bus_ckmr_-xus*yus         100001
 y=bus_ckmr_xus*yus+mr       100010
 y=bus_ckmr_-xus*yus+mr      100011
 y=bus_ckmr_xus*yus-mr       100110
 y=bus_ckmr_-xus*yus-mr      100111
 y=bus_ckmr_xtc*yus          101000
 y=bus_ckmr_-xtc*yus         101001
```

```
 y=bus_ckmr_-xtc*yus+mr      101011
 y=bus_ckmr_xtc*yus-mr       101110
 y=bus_ckmr_-xtc*yus-mr      101111
 y=bus_ckmr_xus*ytc          110000
 y=bus_ckmr_-xus*ytc         110001
 y=bus_ckmr_xus*ytc+mr       110010
 y=bus_ckmr_-xus*ytc+mr      110011
 y=bus_ckmr_xus*ytc-mr      110110
 y=bus_ckmr_-xuc*ytc-mr      110111
 y=bus_ckmr_xtc*ytc          111000
 y=bus_ckmr_-xtc*ytc         111001
 y=bus_ckmr_xtc*ytc+mr      111010
 y=bus_ckmr_-xtc*ytc+mr     111011
 y=bus_ckmr_xtc*ytc-mr       111110
 y=bus_ckmr_-xtc*ytc-mr      111111
 ]
;9
#mac_rnd_pins: FIELD 3-4,WIDTH 2,DEFAULT nornd
  VALUES  [
 rnd14     01
 rnd15     10
 nornd     00
          ]
 ;10
 #mac_control: FIELD 2-2,WIDTH 1,DEFAULT macdisable
   VALUES   [
 macenable  1
 macdisable  0
     ]
 ;11
 #latch_control: FIELD 0-1,WIDTH  2,DEFAULT norwlatch
   VALUES  [
 rdlatch   11
 wrlatch   10
 norwlatch  00
         ]
```

```
;Object file for 1-D convolution

$h
d0050 0001
d0051 0002
d0052 0003
d0053 0004
d0054 0005
p0050 000001421E0005
p0051 00000402988000
p0052 00000542990000
p0053 00000002000203
p0054 003000006000000
p0055 0078016A000000
p0056 00E4004A000000
p0057 00E0000F000000
p0058 00000001601900
p0059 01000163489C40
p005A 00C4001B9C8000
p005B 00000002000403
p005C 00000000000000
p005D 017400017113A4
p005E 01CC015E000000
$$




;Object file for MATRIX multiplication
$h
p0100 00000142604000
p0101 000004021E8000
p0102 00000542990000
p0103 00000012684000
p0104 00000002000203
p0105 003000006000000
```

```
p0108 00E40013088000
p0109 00E0000A800000
p010A 00000001601900
p010B 0100042B689C40
p010C 00000000000000
p010D 00C400020000403
p010E 00000000000000
p010F 017400017113A4
p0110 01CC0426000000
p0111 00C80000828000
p0112 017400018E8000
p0113 01CC0422628000
$$
```

APPENDIX E

```
; THIS DEF. FILE IS FOR THE MICROCODED SYSTEM 2
TITLE MICROSYSTEM
DEFN_FILE "SMACRO1.DAT"
WORD WIDTH 60 BITS
DATA_LOC_WIDTH 16
PROGRAM_LOC_WIDTH  60
;1
#sequencer:FIELD 53-59,WIDTH 7, DEFAULT cont
  OPCODE_BIT_NOTATIONS [
   kk (    ;Conditions
  00  unconditional
  01  notflag
  10  flag
  11  sign
  )
    cc (    ;Selects the relevent register(R3-R0)and
  C0-C3   ;/or counter(C3-C0).
  )
    ii (    ;Decides number to be added incase of
  I0-I3   ; AIRSP instruction.
  )
  ]
   INSTR_NOT_AVAILABLE [
  jrc(sign)(c0)
  jrc(sign)(c1)
  jrc(sign)(c2)
  jrc(sign)(c3)
  branch(sign)(c0)
  branch(sign)(c1)
  branch(sign)(c2)
  branch(sign)(c3)
  ]
   BRANCH_INSTR_WHICH_NEED_DATA [
 ABSOLUTE (
   jsa
```

```
  jda
  jdrst
 )
RELATIVE (
 jsr
 jdr
 )
 ]
  VALUES [
;jump & branch instruction

 jpcof 0010101
 jpcnf 0110101
 jtwo  101kk01
 jda  111kk11
 jdr  111kk01
 jdi  101kk10
 jdrst 10011cc
 jrs  11011cc
 jsa  111kk00
 jsr  111kk10
 rtn  101kk11
 branch 100kkcc

 ;stack operation
 ;subroutine stack

 psdss 0011110
 ppssd 0111110
 wrssp 0001110
 rdssp 0101100
 dssp  0000010

 ;register stack
 sgsp  0000111
 slsp  0000110
 rdrsp 0101111
 wrrsp 0001100
 pspc  0100011
```

```
psgsp 0000101
ppgsp 0000100
psdrs 0011111
pprsd 0111111
airsp 01010ii
sirsp 0001111
s4rsp 0111100

;status register operations:
rdsr  0101110
wrsr   0011100
pssr  0100001
ppsr  0100010

;counter operations:
wrcntr 01110cc
clrs  0010100
sets  0110100
pscntr 00010cc
ppcntr 00110cc
dccntr 01100cc
ifcdec 101kk00

;Interrupt control:
ccir  0010001
cair  0000001
rtnir 0000011
rdiv  0101101
wriv  0001101
irmbc 0010011
irmbs 0010010
disir 0010110
enair 0110110
slir  0010111
stir  0110111
slrivp 0011101

;relative address width controls:
rel16 0100100
```

```
  rel12 0100111
  rel8  0100110

  ;miscellaneous instructions:
  cont  0000000
  idle  0010000
  ihc   0100101
  wcs   0100000


  ]
;2
#data:FIELD 37-52,WIDTH 16,DEFAULT zero

  VALUES [
 zero  0000000000000000
 ]

;3
#DATA_ENABLE:FIELD 36-36,WIDTH 1,DEFAULT disable
        VALUES [
      enable   1
      disable  0
        ]
;4
#address_generator: FIELD 26-35,WIDTH 10,DEFAULT nop

  OPCODE_BIT_NOTATIONS [
   cc (  ;Comparison Register number
  c0-c3
  )
   rrr (  ;Three bit Address register number
  R0-R7
  )
   rrrr ( ;Four bit Address register number
  r0-r15
  )
   bb (  ;Base (offset) register number
  b0-b3
```

```
 )
  ii (  ;Initialisation register number
 i0-i3
 )
  pp (  ;Two bit precision code
 p0-p3
 )
  x (  ;One bit control bit
 x0-x1
 )

 ]
VALUES [
 ;looping instructions

yinc    1011ccrrrr
ydec  1010ccrrrr
yadd  11ccbb1rrr
ysub  11ccbb0rrr

 ;register transfer instructions:
yrtr  000101rrrr
yrtb  0011bbrrrr
yrtc  0010ccrrrr
dti  00001111ii
itr  1000iirrrr
btr  0100bbrrrr
rtd  000100rrrr
ctd  00001100cc
btd  00001101bb
itd  00001110ii

 ;logical and shift instructions
yor  0111bbrrrr
yand  0110bbrrrr
yxor  0101bbrrrr
yasr  000111rrrr
ylsl  000110rrrr
```

```
 rst  0000000001
 dtcr  0000101110
 crtd  0000101111
 seti  0000100iix
 setp  00001010pp
 sety  000001001x
 selr  000001101x
 selb  000001100x
 setu  000001011x
 seta  000001010x

 wra  0000101100
 rda  0000101101
 lda  0000011110

 ydty  0000011111
 yrev  1001bbrrrr
 nop     0000000000
 ]

;5
#ag_gen_pin: FIELD 25-25,WIDTH 1,DEFAULT nodsel

  VALUES [
   nodsel 0
 dsel 1
 ]
;6
#ag_air_select: FIELD 24-24,width 1,default noair
VALUES [
 air 1
 noair 0
 ]

;7
#data_mem_control: FIELD 22-23,WIDTH 2,default norw
   VALUES
 [
  rd    11
```

```
 y=bus_ckmr_-xus*ytc          110001
 y=bus_ckmr_xus*ytc+mr        110010
 y=bus_ckmr_-xus*ytc+mr       110011
 y=bus_ckmr_xus*ytc-mr       110110
 y=bus_ckmr_-xuc*ytc-mr       110111
 y=bus_ckmr_xtc*ytc           111000
 y=bus_ckmr_-xtc*ytc          111001
 y=bus_ckmr_xtc*ytc+mr       111010
 y=bus_ckmr_-xtc*ytc+mr      111011
 y=bus_ckmr_xtc*ytc-mr        111110
 y=bus_ckmr_-xtc*ytc-mr       111111
 ]
;9
#mac_rnd_pins: FIELD 14-15,WIDTH 2,DEFAULT nornd
  VALUES  [
 rnd14      01
 rnd15      10
 nornd      00
           ]

;10
#mac1:FIELD 8-13,WIDTH 6,default nop1
  VALUES [
 nop1            000000
 ckmr1           000100
 x=bus1          001000
 ls=bus1         010000
 ms=bus1         010100
 ex=bus1         010010
 ls=ms11          010001
 ms=ex1          010101
 ex=signextms1     010011
 ms=signextls1     010111
 bus=ex1         001101
 bus=ex(sl)1     001100
 bus=ms1         011001
 bus=ms(sl)1     011000
 bus=ms(sat)1    011011
 bus=ms(slsat)1    011010
```

```
  wr     10
  norw   00
         ]
;8
#mac:FIELD 16-21,WIDTH 6,default nop
  VALUES [
 nop             000000
 ckmr            000100
 x=bus           001000
 ls=bus          010000
 ms=bus          010100
 ex=bus          010010
 ls=ms           010001
 ms=ex           010101
 ex=signextms     010011
 ms=signextls     010111
 bus=ex          001101
 bus=ex(sl)      001100
 bus=ms          011001
 bus=ms(sl)      011000
 bus=ms(sat)     011011
 bus=ms(slsat)    011010
 bus=ls          011101
 bus=ls(sl)      011100
 bus=ls(sat)     011111
 bus=ls(slsat)    011110
 y=bus_ckmr_xus*yus          100000
 y=bus_ckmr_-xus*yus         100001
 y=bus_ckmr_xus*yus+mr       100010
 y=bus_ckmr_-xus*yus+mr      100011
 y=bus_ckmr_xus*yus-mr       100110
 y=bus_ckmr_-xus*yus-mr      100111
 y=bus_ckmr_xtc*yus          101000
 y=bus_ckmr_-xtc*yus         101001
 y=bus_ckmr_xtc*yus+mr       101010
 y=bus_ckmr_-xtc*yus+mr      101011
 y=bus_ckmr_xtc*yus-mr       101110
 y=bus_ckmr_-xtc*yus-mr      101111
 y=bus_ckmr_xus*ytc          110000
```

```
 bus=ls1        011101
 bus=ls(sl)1    011100
 bus=ls(sat)1   011111
 bus=ls(slsat)1   011110
 y=bus_ckmr_xus*yus1          100000
 y=bus_ckmr_-xus*yus1         100001
 y=bus_ckmr_xus*yus+mr1       100010
 y=bus_ckmr_-xus*yus+mr1      100011
 y=bus_ckmr_xus*yus-mr1       100110
 y=bus_ckmr_-xus*yus-mr1      100111
 y=bus_ckmr_xtc*yus1          101000
 y=bus_ckmr_-xtc*yus1         101001
 y=bus_ckmr_xtc*yus+mr1       101010
 y=bus_ckmr_-xtc*yus+mr1      101011
 y=bus_ckmr_xtc*yus-mr1       101110
 y=bus_ckmr_-xtc*yus-mr1      101111
 y=bus_ckmr_xus*ytc1          110000
 y=bus_ckmr_-xus*ytc1         110001
 y=bus_ckmr_xus*ytc+mr1       110010
 y=bus_ckmr_-xus*ytc+mr1      110011
 y=bus_ckmr_xus*ytc-mr1      110110
 y=bus_ckmr_-xuc*ytc-mr1      110111
 y=bus_ckmr_xtc*ytc1          111000
 y=bus_ckmr_-xtc*ytc1         111001
 y=bus_ckmr_xtc*ytc+mr1      111010
 y=bus_ckmr_-xtc*ytc+mr1     111011
 y=bus_ckmr_xtc*ytc-mr1       111110
 y=bus_ckmr_-xtc*ytc-mr1      111111
 ]
;11
#mac_rnd_pins1: FIELD 6-7,WIDTH 2,DEFAULT norndi

  VALUES  [
 rnd141      01
 rnd151      10
 norndi     00
           ]
;12
#mac_control: FIELD 5-5,WIDTH 1,DEFAULT macdisable
```

```
    VALUES [
 macenable  1
 macdisable  0
           ]
;13
#mac1_control: FIELD 4-4,WIDTH 1,DEFAULT mac1disable
    VALUES [
 mac1enable  1
 mac1disable  0
    ]
;14
#latch_control: FIELD 2-3,WIDTH 2,DEFAULT norwlatch
    VALUES [
 rdlatch   11
 wrlatch   10
 norwlatch 00
       ]
;15
#latch1_control: FIELD  0-1,WIDTH  2,DEFAULT  norwlatch1
    VALUES [
 rdlatch1   11
 wrlatch1   10
 norwlatch1  00
      ]
```

```
 ;Object file for 1-D convoluton

$h
p0050 00000A10F0000000
p0051 00000A9302000000
p0052 00001FB4C8000000
p0053 000024B4D0000000
p0054 00004014D8000000
p0055 00003FF4DC000000
p0056 0000001000101000F
```

```
p0057 0180003000000000
p0058 03C00C3000000000
p0059 03C00DD000000000
p005A 076000D000000000
p005B 0740005414000000
p005C 0700005800000000
p005D 0000000B00C80000
p005E 0000000A10E20000
p005F 0000000B48C00803
p0060 08000BBA14C02203
p0061 064000BCF0000000
p0062 0000003000040B0F
p0063 0000000001D2223
p0064 0000000000000000
p0065 000000141420200F
p0066 0000007CC8000000
p0067 0BA0000BD8B01D12
p0068 0E600B9000000000
p0069 0000000A08000000
p006A 0720007800000000
p006B 0000000B00C80803
p006C 0000000A10E20000
p006D 08200D7A08C02203
p006E 066000DCF0000000
p006F 000000100020200F
p0070 0000009CE8000000
p0071 0000000B189D0020
p0072 0BA0000A1C801D12
p0073 0E600D50000000000
$$

;Object file for MATRIX multiplication
$h
p0050 00000A1302000001
p0051 00002010F4000000
p0052 00002090F8000000
p0053 00002A14CC000000
p0054 000000100010100F
p0055 0180003000000000
```

```
p0056 03C00BD000000000
p0057 0740005000000000
p0058 0720003844000000
p0059 0000000888000000
p005A 0700005400000000
p005B 0000000B00C80803
p005C 0000000B44E20000
p005D 08000B7E88C02203
p005E 0000000000000000
p005F 000000100020200F
p0060 0000009EE4000000
p0061 0000000E8C9D0020
p0062 0620000B8C801D12
p0063 0BA0009EE8000000
p0064 0E600E50000000000
p0065 0640000414000000
p0066 0BA0009CF4000000
p0067 0E600B1314000000
$$
```

APPENDIX F

```
; THIS DEF. FILE IS FOR THE MICROCODED SYSTEM 3
TITLE MICROSYSTEM
DEFN_FILE "SMACRO2.DAT"
WORD WIDTH 74 BITS
DATA_LOC_WIDTH 16
PROGRAM_LOC_WIDTH  74
;1
#sequencer:FIELD 67-73,WIDTH 7, DEFAULT cont
  OPCODE_BIT_NOTATIONS [
   kk (    ;Conditions
  00  unconditional
  01  notflag
  10  flag
  11  sign
  )
    cc (    ;Selects the relevent register(R3-R0)and
  C0-C3   ;/or counter(C3-C0).
  )
    ii (    ;Decides number to be added incase of
  I0-I3   ; AIRSP instruction.
  )
 ]
  INSTR_NOT_AVAILABLE [
 jrc(sign)(c0)
 jrc(sign)(c1)
 jrc(sign)(c2)
 jrc(sign)(c3)
 branch(sign)(c0)
 branch(sign)(c1)
 branch(sign)(c2)
 branch(sign)(c3)
  ]
   BRANCH_INSTR_WHICH_NEED_DATA [
 ABSOLUTE (
 jsa
```

```
jda
jdrst
 )
RELATIVE (
jsr
jdr
 )
 ]
 VALUES [
;jump & branch instruction

jpcof 0010101
jpcnf 0110101
jtwo  101kk01
jda   111kk11
jdr   111kk01
jdi   101kk10
jdrst 10011cc
jrs   11011cc
jsa   111kk00
jsr   111kk10
rtn   101kk11
branch 100kkcc

;stack operation
;subroutine stack

psdss 0011110
ppssd 0111110
wrssp 0001110
rdssp 0101100
dssp  0000010

;register stack
sgsp  0000111
slsp  0000110
rdrsp 0101111
wrrsp 0001100
pspc  0100011
```

```
psgsp 0000101
ppgsp 0000100
psdrs 0011111
pprsd 0111111
airsp 01010ii
sirsp 0001111
s4rsp 0111100

;status register operations:
rdsr  0101110
wrsr   0011100
pssr  0100001
ppsr  0100010

;counter operations:
wrcntr 01110cc
clrs  0010100
sets  0110100
pscntr 00010cc
ppcntr 00110cc
dccntr 01100cc
ifcdec 101kk00

;Interrupt control:
ccir  0010001
cair  0000001
rtnir 0000011
rdiv  0101101
wriv  0001101
irmbc 0010011
irmbs 0010010
disir 0010110
enair 0110110
slir  0010111
stir  0110111
slrivp 0011101

;relative address width controls:
rel16 0100100
```

```
rel12 0100111
rel8  0100110

 ;miscellaneous instructions:
cont  0000000
idle  0010000
ihc   0100101
wcs   0100000


 ]
;2
#data:FIELD 51-66,WIDTH 16,DEFAULT zero

 VALUES [
zero  0000000000000000
]

;3
#DATA_ENABLE:FIELD 50-50,WIDTH 1,DEFAULT disable
       VALUES [
enable   1
disable  0
       ]
;4
#address_generator: FIELD 40-49,WIDTH 10,DEFAULT nop

 OPCODE_BIT_NOTATIONS [
  cc (  ;Comparison Register number
 c0-c3
 )
  rrr (  ;Three bit Address register number
 R0-R7
 )
  rrrr ( ;Four bit Address register number
 r0-r15
 )
  bb (  ;Base (offset) register number
 b0-b3
```

```
 )
  ii (  ;Initialisation register number
 i0-i3
 )
  pp (  ;Two bit precision code
 p0-p3
 )
  x (  ;One bit control bit
 x0-x1
 )

 ]
VALUES [
 ;looping instructions

yinc    1011ccrrrr
ydec  1010ccrrrr
yadd  11ccbb1rrr
ysub  11ccbb0rrr

 ;register transfer instructions:
yrtr  000101rrrr
yrtb  0011bbrrrr
yrtc  0010ccrrrr
dti  00001111ii
itr  1000iirrrr
btr  0100bbrrrr
rtd  000100rrrr
ctd  00001100cc
btd  00001101bb
itd  00001110ii

 ;logical and shift instructions
yor  0111bbrrrr
yand  0110bbrrrr
yxor  0101bbrrrr
yasr  000111rrrr
ylsl  000110rrrr
```

```
 rst  0000000001
 dtcr  0000101110
 crtd  0000101111
 seti  000010Oiix
 setp  00001010pp
 sety  000001001x
 selr  000001101x
 selb  000001100x
 setu  000001011x
 seta  000001010x

 wra  0000101100
 rda  0000101101
 lda  0000011110

 ydty  0000011111
 yrev  1001bbrrrr
 nop     0000000000
 ]

;5
#ag_gen_pin: FIELD 39-39,WIDTH 1,DEFAULT nodsel

  VALUES [
 nodsel 0
 dsel 1
 ]
;6
#ag_air_select: FIELD 38-38,width 1,default noair
VALUES [
 air 1
 noair 0
 ]

;7
#data_mem_control: FIELD 36-37,WIDTH 2,default norw
   VALUES
 [
 rd    11
```

```
 wr     10
 norw   00
        ]
;8
#mac:FIELD 30-35,WIDTH 6,default nop
  VALUES [
 nop            000000
 ckmr           000100
 x=bus          001000
 ls=bus         010000
 ms=bus         010100
 ex=bus         010010
 ls=ms          010001
 ms=ex          010101
 ex=signextms     010011
 ms=signextls     010111
 bus=ex         001101
 bus=ex(sl)     001100
 bus=ms         011001
 bus=ms(sl)     011000
 bus=ms(sat)    011011
 bus=ms(slsat)    011010
 bus=ls         011101
 bus=ls(sl)     011100
 bus=ls(sat)    011111
 bus=ls(slsat)    011110
 y=bus_ckmr_xus*yus          100000
 y=bus_ckmr_-xus*yus         100001
 y=bus_ckmr_xus*yus+mr       100010
 y=bus_ckmr_-xus*yus+mr      100011
 y=bus_ckmr_xus*yus-mr       100110
 y=bus_ckmr_-xus*yus-mr      100111
 y=bus_ckmr_xtc*yus          101000
 y=bus_ckmr_-xtc*yus         101001
 y=bus_ckmr_xtc*yus+mr       101010
 y=bus_ckmr_-xtc*yus+mr      101011
 y=bus_ckmr_xtc*yus-mr       101110
 y=bus_ckmr_-xtc*yus-mr      101111
 y=bus_ckmr_xus*ytc          110000
```

```
 y=bus_ckmr_xus*ytc+mr       110010
 y=bus_ckmr_-xus*ytc+mr      110011
 y=bus_ckmr_xus*ytc-mr      110110
 y=bus_ckmr_-xuc*ytc-mr      110111
 y=bus_ckmr_xtc*ytc          111000
 y=bus_ckmr_-xtc*ytc         111001
 y=bus_ckmr_xtc*ytc+mr      111010
 y=bus_ckmr_-xtc*ytc+mr     111011
 y=bus_ckmr_xtc*ytc-mr       111110
 y=bus_ckmr_-xtc*ytc-mr      111111
 ]
;9
#mac_rnd_pins: FIELD 28-29,WIDTH 2,DEFAULT nornd
  VALUES  [
 rnd14      01
 rnd15      10
 nornd      00
        ]

;10
#address_generator1: FIELD 18-27,WIDTH 10,DEFAULT nop1

  OPCODE_BIT_NOTATIONS [
   cc (  ;Comparison Register number
  c0-c3
  )
   rrr (  ;Three bit Address register number
  R0-R7
  )
   rrrr ( ;Four bit Address register number
  r0-r15
  )
   bb (  ;Base (offset) register number
  b0-b3
  )
   ii (  ;Initialisation register number
  i0-i3
  )
```

```
 pp (  ;Two bit precision code
p0-p3
)
 x (  ;One bit control bit
x0-x1
)

]
VALUES [
 ;looping instructions

yinc1    1011ccrrrr
ydec1  1010ccrrrr
yadd1  11ccbb1rrr
ysub1  11ccbb0rrr

 ;register transfer instructions:
yrtr1  000101rrrr
yrtb1  0011bbrrrr
yrtc1  0010ccrrrr
dti1  00001111ii
itr1  1000iirrrr
btr1  0100bbrrrr
rtd1  000100rrrr
ctd1  00001100cc
btd1  00001101bb
itd1  00001110ii

 ;logical and shift instructions
yor1  0111bbrrrr
yand1  0110bbrrrr
yxor1  0101bbrrrr
yasr1  000111rrrr
ylsl  000110rrrr

rst1  0000000001
dtcr1  0000101110
crtd1  0000101111
seti1  0000100iix
```

```
 setp1  00001010pp
 sety1  000001001x
 selr1  000001101x
 selb1  000001100x
 setu1  000001011x
 seta1  000001010x

 wra1  0000101100
 rda1  0000101101
 lda1  0000011110

 ydty1  0000011111
 yrev1  1001bbrrrr
 nop1    0000000000
 ]

;11
#ag1_gen_pin: FIELD 17-17,WIDTH 1,DEFAULT nodsel1

  VALUES [
 nodsel1 0
 dsel1 1
 ]
;12
#ag1_air_select: FIELD 16-16,width 1,default noair1
VALUES [
 air1 1
 noair1 0
 ]

;13
#data1_mem_control: FIELD 14-15,WIDTH 2,default norw1
   VALUES
 [
 rd1    11
 wr1    10
 norw1  00
        ]
;14
```

```
#mac1:FIELD 8-13,WIDTH 6,default nop1
 VALUES [
nop1            000000
ckmr1           000100
x=bus1          001000
ls=bus1         010000
ms=bus1        010100
ex=bus1         010010
ls=ms1         010001
ms=ex1          010101
ex=signextms1     010011
ms=signextls1     010111
bus=ex1         001101
bus=ex(sl)1     001100
bus=ms1         011001
bus=ms(sl)1     011000
bus=ms(sat)1    011011
bus=ms(slsat)1    011010
bus=ls1         011101
bus=ls(sl)1     011100
bus=ls(sat)1    011111
bus=ls(slsat)1    011110
y=bus_ckmr_xus*yus1           100000
y=bus_ckmr_-xus*yus1          100001
y=bus_ckmr_xus*yus+mr1        100010
y=bus_ckmr_-xus*yus+mr1       100011
y=bus_ckmr_xus*yus-mr1        100110
y=bus_ckmr_-xus*yus-mr1       100111
y=bus_ckmr_xtc*yus1          101000
y=bus_ckmr_-xtc*yus1         101001
y=bus_ckmr_xtc*yus+mr1        101010
y=bus_ckmr_-xtc*yus+mr1       101011
y=bus_ckmr_xtc*yus-mr1        101110
y=bus_ckmr_-xtc*yus-mr1       101111
y=bus_ckmr_xus*ytc1           110000
y=bus_ckmr_-xus*ytc1          110001
y=bus_ckmr_xus*ytc+mr1        110010
y=bus_ckmr_-xus*ytc+mr1       110011
y=bus_ckmr_xus*ytc-mr1       110110
```

```
 y=bus_ckmr_-xuc*ytc-mr1      110111
 y=bus_ckmr_xtc*ytc1         111000
 y=bus_ckmr_-xtc*ytc1         111001
 y=bus_ckmr_xtc*ytc+mr1      111010
 y=bus_ckmr_-xtc*ytc+mr1     111011
 y=bus_ckmr_xtc*ytc-mr1       111110
 y=bus_ckmr_-xtc*ytc-mr1      111111
 ]
;15
#mac1_rnd_pins: FIELD 6-7,WIDTH 2,DEFAULT norndl
  VALUES  [
 rndl41      01
 rndl51      10
 norndl      00
          ]
;16
#latch_control: FIELD 4-5,WIDTH 2,DEFAULT norwlatch
   VALUES [
 rdlatch      11
 wrlatch      10
 norwlatch    00
        ]
;17
#latch1_control: FIELD 2-3,WIDTH 2,DEFAULT norwlatch1
   VALUES [
 rdlatch1     11
 wrlatch1     10
 norwlatch1   00
          ]
;18
#mac_control: FIELD 1-1,WIDTH 1,DEFAULT macdisable
   VALUES [
 macenable   1
 macdisable   0
          ]
;19
#mac1_control: FIELD 0-0,WIDTH 1,DEFAULT mac1disable
   VALUES  [
 mac1enable   1
```

```
mac1disable 0
          ]
```

```
;Object file for MATRIX multiplication

$h
p0100 00000284C08003020000
p0101 00000B043D0000F40000
p0102 00000A84000004C80000
p0103 00000AB4000004CC0000
p0104 0000000400040000103C
p0105 0060000C0000000000000
p0106 00F0086C0000000000000
p0107 00F00894000000000000
p0108 01D000040000000000000
p0109 01C80026110008440000
p010A 01C000150000040000000
p010B 00000002C0320B00C800
p010C 0200085EC1388E04E200
p010D 00000000000000000000
p010E 0000000400080000203C
p010F 00000000000000000000
p0110 0000000000074B48802E
p0111 02080854000000B4C9D01
p0112 00000034000000CE80000
p0113 00000034000000CEC0000
p0114 0210084CC0000030000000
$$

;Object file for 1-D convolution
$h
p0100 000002843C0000F00000
p0101 000002A4C08000000000
p0102 000007ED320000000000
p0103 0000092C000004D00000
```

```
p0104 00000A84000004DB0000
p0105 00000A7C000004DC0000
p0106 0000000400040000103C
p0107 0060000C0000000000000
p0108 00F0087C0000000000000
p0109 00F008DC0000000000000
p010A 01D80034000000000000
p010B 01D0001400000000000
p010C 01C000150000080000000
p010D 00000002C2320B00C800
p010E 0200086E80388A10E200
p010F 0000001F3200000000030
p0110 0000000C0001000000800
p0111 000000000007400022 2E
p0112 0000002D00000CF00000
p0113 019000040008000203C
p0114 00000000000000000000
p0115 02E800000000B989D01
p0116 039808640000000000000
p0117 0000000282000000000
p0118 01C8001E00000B000000
p0119 00000002C0320B00C800
p011A 020808CE82388A10E200
p011B 000000273A0000000000
p011C 0000000400080000203C
p011D 019800340000CF00000
p011E 0000000000074A1C802E
p011F 02E800000000B189D01
p0120 039808C40000000000000
$$
```